O'REILLY®



图灵程序设计丛书

Python科学计算最佳实践

SciPy指南

Elegant SciPy

[澳] 胡安·努内兹-伊格莱西亚斯
[美] 斯特凡·范德瓦尔特 [澳] 哈丽雅特·达士诺 著

陈光欣 译

 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

陈光欣

毕业于清华大学并留校工作，主要兴趣为数据分析与数据挖掘。

数字版权声明



图灵程序设计丛书

Python科学计算最佳实践：SciPy指南

Elegant SciPy

The Art of Scientific Python

[澳] 胡安·努内兹-伊格莱西亚斯
[美] 斯特凡·范德瓦尔特　著
[澳] 哈丽雅特·达士诺
陈光欣　译

Beijing • Boston • Farnham • Sebastopol • Tokyo　O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

Python科学计算最佳实践 : SciPy指南 / (澳) 胡安·努内兹-伊格莱西亚斯, (美) 斯特凡·范德瓦尔特, (澳) 哈丽雅特·达士诺著 ; 陈光欣译. -- 北京 : 人民邮电出版社, 2019.1
（图灵程序设计丛书）
ISBN 978-7-115-49912-7

Ⅰ. ①P… Ⅱ. ①胡… ②斯… ③哈… ④陈… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第250525号

内 容 提 要

本书旨在介绍开源的 Python 算法库和数学工具包 SciPy。近年来，基于 NumPy 和 SciPy 的完整生态系统迅速发展起来，并在天文学、生物学、气象学和气候科学，以及材料科学等多个学科得到了广泛应用。本书结合大量代码实例，详尽展示了 SciPy 的强大科学计算能力，包括用 NumPy 和 SciPy 进行分位数标准化，用 ndimage 实现图像区域网络，频率与快速傅里叶变换，用稀疏坐标矩阵实现列联表，SciPy 中的线性代数，SciPy 中的函数优化等。

本书适合 Python 程序员以及计算科学领域从业人员阅读参考。

◆ 著　[澳] 胡安·努内兹-伊格莱西亚斯
[美] 斯特凡·范德瓦尔特
[澳] 哈丽雅特·达士诺
译　陈光欣
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：14.25
字数：350千字　2019年 1 月第 1 版
印数：1 – 3 500册　2019年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-7362号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言······ix

第 1 章　优雅的 NumPy：Python 科学应用的基础······1

1.1　数据简介：什么是基因表达······2
1.2　NumPy 的 *N* 维数组······6
1.2.1　为什么用 *N* 维数组代替 Python 列表······7
1.2.2　向量化······9
1.2.3　广播······9
1.3　探索基因表达数据集······10
1.4　标准化······13
1.4.1　样本间的标准化······13
1.4.2　基因间的标准化······19
1.4.3　样本与基因标准化：RPKM······21
1.5　小结······27

第 2 章　用 NumPy 和 SciPy 进行分位数标准化······28

2.1　获取数据······30
2.2　独立样本间的基因表达分布差异······30
2.3　计数数据的双向聚类······33
2.4　簇的可视化······35
2.5　预测幸存者······37
2.5.1　进一步工作：使用 TCGA 患者簇······41
2.5.2　进一步工作：重新生成 TCGA 簇······41

第 3 章　用 `ndimage` 实现图像区域网络······42

3.1　图像就是 NumPy 数组······43
3.2　信号处理中的滤波器······48

3.3 图像滤波（二维滤波器）53
3.4 通用滤波器：邻近值的任意函数 55
3.4.1 练习：康威的生命游戏 56
3.4.2 练习：Sobel 梯度幅值 56
3.5 图与 NetworkX 库 57
3.6 区域邻接图 60
3.7 优雅的 `ndimage`：如何根据图像区域建立图对象 63
3.8 归纳总结：平均颜色分割 65

第 4 章 频率与快速傅里叶变换 67

4.1 频率的引入 67
4.2 示例：鸟鸣声谱图 69
4.3 历史 74
4.4 实现 75
4.5 选择离散傅里叶变换的长度 75
4.6 更多离散傅里叶变换概念 77
4.6.1 频率及其排序 77
4.6.2 加窗 83
4.7 实际应用：分析雷达数据 86
4.7.1 频域中的信号性质 91
4.7.2 加窗之后 93
4.7.3 雷达图像 95
4.7.4 快速傅里叶变换的进一步应用 99
4.7.5 更多阅读 99
4.7.6 练习：图像卷积 100

第 5 章 用稀疏坐标矩阵实现列联表 101

5.1 列联表 102
5.1.1 练习：混淆矩阵的计算复杂度 103
5.1.2 练习：计算混淆矩阵的另一种方法 103
5.1.3 练习：多类混淆矩阵 104
5.2 `scipy.sparse` 数据格式 104
5.2.1 COO 格式 104
5.2.2 练习：COO 表示 105
5.2.3 稀疏行压缩格式 106
5.3 稀疏矩阵应用：图像转换 108
5.4 回到列联表 112
5.5 图像分割中的列联表 113
5.6 信息论简介 114
5.7 图像分割中的信息论：信息变异 117
5.8 转换 NumPy 数组代码以使用稀疏矩阵 119
5.9 使用信息变异 120

第 6 章　SciPy 中的线性代数 …… 128

6.1　线性代数基础 …… 128
6.2　图的拉普拉斯矩阵 …… 129
6.3　大脑数据的拉普拉斯矩阵 …… 134
6.3.1　练习：显示近邻视图 …… 138
6.3.2　练习挑战：稀疏矩阵线性代数 …… 138
6.4　PageRank：用于声望和重要性的线性代数 …… 139
6.4.1　练习：处理悬挂节点 …… 144
6.4.2　练习：不同特征向量方法的等价性 …… 144
6.5　结束语 …… 144

第 7 章　SciPy 中的函数优化 …… 145

7.1　SciPy 优化模块：`sicpy.optimize` …… 146
7.2　用 `optimize` 进行图像配准 …… 152
7.3　用 basin hopping 算法避开局部最小值 …… 155
7.4　选择正确的目标函数 …… 156

第 8 章　用 Toolz 在笔记本电脑上玩转大数据 …… 163

8.1　用 `yield` 进行流处理 …… 164
8.2　引入 Toolz 流库 …… 167
8.3　*k*-mer 计数与错误修正 …… 169
8.4　柯里化：流的调料 …… 173
8.5　回到 *k*-mer 计数 …… 175
8.6　全基因组的马尔可夫模型 …… 177

后记 …… 182

附录　练习答案 …… 186

作者简介 …… 206

封面简介 …… 206

前言

> 与那种老式刻板的结婚礼服不同，如果用技术术语来描述的话，它是优雅的，宛如通过寥寥几行代码就能得出令人赞叹的结果的计算机算法。
>
> ——格雷姆·辛浦生，《罗茜效应》

欢迎阅读本书。因为我们将用大部分篇幅讨论书名中的“SciPy”，所以先花点时间说明一下“优雅”这个词。现在有很多介绍 SciPy 库的手册、教程和文档网站，但本书讲述得更加深入，它不但会教你如何编写有效的代码，还会激励你将代码变得更加酷炫！

在《罗茜效应》（一本妙趣横生的小说；学完本书后，你可以阅读一下它的前作《罗茜计划》）中，格雷姆·辛浦生颠覆了“优雅”这个词的传统含义。多数人会用这个词来形容某物在视觉上的简单、时髦和优美，比如第一代 iPhone。但格雷姆·辛浦生著作中的主人公唐·蒂尔曼却用计算机算法来**定义**优雅。读完本书后，希望你能确切地理解他的意思，因为你将在本书中阅读或编写一段优雅的代码，并在它美妙高雅的光辉下感受内心的平静。

一段良好的代码在**感觉上**就是正确的。当查看这种代码时，它的意图是**明确的**，形式是**简洁的**（但不至于晦涩），而且能**高效**地完成当前工作。对于作者而言，分析优雅代码的乐趣在于找出其中隐藏的知识，以及由其激发出的解决新编程问题时的**创造性**。

具有讽刺意味的是，创造性还会引诱我们为了炫耀自己的睿智而编写出难以理解的代码，而承受这种代价的是代码阅读者。PEP 8（“Python 编码规范”）和 PEP 20（“Python 之禅”）提醒我们，阅读代码要比编写代码频繁得多，因此可读性最重要。

优雅代码的简洁性来自抽象和正确使用函数，而非大量的函数嵌套调用。可能需要一两分钟来领会这种代码，但最终肯定有一个恍然大悟的时刻。一旦搞清楚代码的各个组成部分，它的正确性就显而易见了。要想提高代码的可读性，可以使用清晰明确的变量和函数名称，并精心编写注释，注释不仅要**描述**代码，还应该**解释**代码。

软件工程师 J. Bradford Hipps 最近在《纽约时报》上发表了一篇文章，他认为，“要想写出更好的代码，就应该阅读一下弗吉尼亚·伍尔芙的作品”。文章选段如下：

> 在实践中，软件开发更多的时候是一种创造性活动，绝不是机械的算法。
>
> 开发人员站在源代码编辑器前，就像是作家面对着空白的稿纸。……他们都对循规蹈矩的做事方式表现出理所当然的焦躁，并从内心深处渴望打破常规。当完成一段程序或一页作品时，它们的质量评判标准在很大程度上是一样的：优雅、简洁、内聚，没有一点故弄玄虚的东西，甚至堪称漂亮。

这也是本书所持的立场。

至此，本书英文版书名中的“优雅”（elegant）讨论完毕，接下来将介绍其中的“SciPy”。

根据上下文，“SciPy”可以是一个软件库、一种生态系统或一个社区。SciPy 如此优秀的部分原因就在于它有特别出色的在线文档和教程，这使得我们根本不需要其他参考书。但是，我们希望本书能够帮助你用 SciPy 编写出最好的代码。

我们选择的代码集中体现了 NumPy、SciPy 和相关库中的高级功能，并演示了这些功能聪明而又优雅的使用方法。初学者可以学会如何通过优美的代码应用这些库来解决实际问题。我们也将用真实的科学数据使书中的示例更加生动有趣。

像 SciPy 本身一样，我们也希望本书是由社区驱动的。书中的很多示例都取自 Python 科学生态系统中的实际代码，它们生动地体现了我们描绘的优雅代码的准则。

目标读者

本书旨在将你的 Python 水平提高到一个新的层次。你将通过实例用最棒的代码来学习 SciPy。

在开始学习之前，你至少应该了解 Python，知道变量、函数、循环和一点 NumPy 的知识。或许你已经通过一些高级读物提高了 Python 技能，比如《流畅的 Python》[1]。如果这不符合你的情况，那么在继续学习本书前，你应该先学习一下 Python 的入门教程，比如 Software Carpentry 的网站。

或许你还搞不清 SciPy stack 到底是一个库还是 IHOP 松饼屋菜单上的一道菜，也不确定什么是 SciPy 最佳实践。或许你是一位科研人员，在网上读过一些 Python 教程，从另一个实验室或自己实验室的前成员那里下载了一些分析脚本，头昏脑胀地乱改了一通。可能你觉得自己正在学习 SciPy 编程的路上孤军奋战，但其实并不是。

在学习过程中，我们会教你如何将互联网作为参考，并介绍一些邮件列表、库和相关会议，在那里你会遇到一些志同道合的科研人员，他们已经在这条路上先行了一步。

虽然这本书你可能只会阅读一次，但也许会多次借此寻求灵感和启发（也可能欣赏一些优雅的代码片段）。

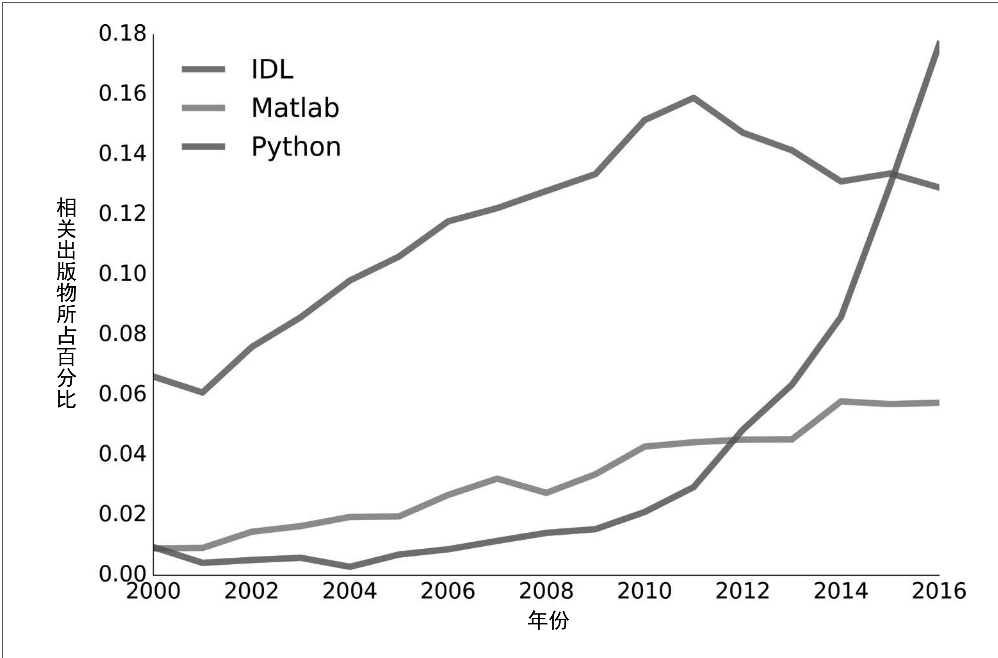
为什么使用SciPy

NumPy 和 SciPy 共同组成了 Python 科学应用生态系统的核心。SciPy 软件库实现了很多用于科学数据处理的函数，比如用于统计学、信号处理、图像处理和函数优化。SciPy 是建

注 1：参见图灵社区：http://www.ituring.com.cn/book/1564。——编者注

立在 NumPy 之上的，NumPy 是 Python 中用于数值型数组计算的库。在过去的几年中，基于 NumPy 和 SciPy，一个包括应用和库文件的完整生态系统迅速发展起来，并在天文学、生物学、气象学和气候科学，以及材料科学等多个学科得到了广泛应用。

这种发展方兴未艾。2014 年，Thomas Robitaille 和 Chris Beaumont 整理了 Python 在天文学领域不断增长的应用。这是我们找到的 2016 年下半年的最新图形结果。



很明显，SciPy 及其相关的库在未来几年将支持大量科学数据分析任务。

再举个例子，Software Carpentry 是一个专门向科研人员传授计算技能的组织，其中最常用的语言就是 Python，现在相关课程已经供不应求。

什么是SciPy生态系统

> SciPy（读作 Sigh Pie）是一个基于 Python 的用于数学、科学和工程学的开源软件生态系统。
>
> ——http://www.scipy.org

SciPy 生态系统是 Python 包的一个松散集合。本书将介绍以下主要内容。

- NumPy 是 Python 科学计算的基础。它提供了高效的数值数组，并广泛支持数值计算，其中包括线性代数、随机数和傅里叶变换。NumPy 的杀手级特性是"*N* 维数组"，又称 ndarray。这些数据结构可以高效地存储数值型数据，并能定义任何维度的网格（稍后将做更多介绍）。
- SciPy 库是一个高效的数值算法集合，用于信号处理、集成、优化和统计学等领域。其中的程序都使用了用户友好型界面进行包装。

- Matplotlib 是一个功能强大的二维（和基本的三维）绘图包。它的名称来自于受 Matlab 启发而发明的语法。
- IPython 是一个交互式的 Python 界面，它允许你快速地与数据和测试方案交互。
- Jupyter 笔记本在浏览器中运行，可以构建样式丰富的文档，将代码、文本、数学表达式和可交互部件组合在一起。[2] 实际上，为了生成本书，需要将文本转换为 Jupyter 笔记本并运行（这样我们就可以知道所有示例都能够正确运行）。Jupyter 最初是作为 IPython 的扩展而开发的，但现在可以支持多种语言，其中包括 Cython、Julia、R、Octave、Bash、Perl 和 Ruby。
- pandas 通过一个便于使用的软件包提供了快速、列式的数据结构。它特别适合处理有标记的数据集，如表格和关系数据库，还适合管理时间序列数据和滑动窗口。pandas 中还有很多便利的数据分析工具，可以解析、清洗、聚合和绘制数据。
- scikit-learn 为机器学习算法提供了统一的接口。
- scikit-image 提供了能与 SciPy 生态系统其他部分完美集成的图像分析工具。

很多 Python 软件包组成了 SciPy 生态系统的其他部分，本书也会涉及其中一些。虽然本书重点介绍 NumPy 和 SciPy，但正是因为存在大量与它们紧密关联的软件包，Python 才成为了科学计算的强大工具。

天翻地覆：从Python 2到Python 3

在使用 Python 的过程中，你应该对哪个 Python 版本更好的争论有所耳闻。或许你对此非常诧异，难道不是最新版本更好吗？（先剧透一下：确实是最新版本更好。）

2008 年末，Python 核心开发团队发布了 Python 3，对这种语言进行了一次重大更新，主要改进包括更好的 Unicode（国际标准）文本处理、类型的一致性以及流式数据处理等。正如 Douglas Adams 对宇宙初创的风趣评论一样 [3]，“这令很多人勃然大怒，被普遍认为是一种倒退”。这是因为，如果不进行一些修改，Python 2.6 或 Python 2.7 的代码一般不能由 Python 3 直接解释（尽管这些改动一般没有什么破坏性）。

向前发展与向后兼容总是要进行一番角力。对于这个问题，Python 核心团队决定做一个彻底的改变，消除 Python（特别是底层 C 语言 API）中的不一致性，从而使 Python 成为一门 21 世纪的语言（Python 1.0 发布于 1994 年，距今已有 20 余年，在技术界已经是相当长的时间）。

以下是 Python 3 中的一处改进。

```
print "Hello World!"   # Python 2 打印语句
print("Hello World!")  # Python 3 打印函数
```

为什么非要不嫌麻烦地加上一对括号呢？是的，括号确实比以前麻烦，但如果你想要输出到另一个**流**中，如常用于存放调试信息的**标准错误流**，该怎么办呢？

注 2：参见 Fernando Perez 的文章“‘Literate computing’ and computational reproducibility: IPython in the age of data-driven journalism”。

注 3：ADAMS D. The Hitchhiker’s guide to the galaxy [M]. London: Pan Books, 1979.

```
print >>sys.stderr, "fatal error"  # Python 2
print("fatal error", file=sys.stderr)  # Python 3
```

此时这种改动看起来就很有价值。Python 2 中的代码到底表示什么意思呢？我们确实不大明白。

Python 3 的另一项改进是将整数除法修改为符合大多数人习惯的形式。（注意：>>> 表示我们正在 Python 交互式环境中进行输入。）

```
# Python 2
>>> 5 / 2
2
# Python 3
>>> 5 / 2
2.5
```

2015 年，Python 3.5 引入的新**矩阵乘法操作符** @ 也非常令人激动，你将在第 5 章和第 6 章中看到这个操作符的实际应用。

Python 3 中的最大改进可能就是对 Unicode 的支持，Unicode 是一种文本编码方式，它不但包括英文字母表，还包括世界上所有的字母表。Python 2 允许你定义一个 Unicode 字符串，如下所示。

```
beta = u"β"
```

但在 Python 3 中，**一切**都是 Unicode。

```
β = 0.5
print(2 * β)

1.0
```

Python 核心团队做出了正确的决策，在 Python 代码中一视同仁地支持所有语言的字符。如今这个决策看起来尤其英明，因为大多数新程序员来自于非英语国家。考虑到互操作性，我们还是建议在多数代码中使用英文字符，但这种能力迟早会派上用场，比如在包含大量数学公式的 Jupyter 笔记本中。



在 IPython 终端或 Jupyter 笔记本中，先输入一个 LaTeX 符号名称，紧接着按 Tab 键，就可以将其转换为 Unicode。例如，`\bet<TAB>` 可以变为 β 。

Python 3 的更新也破坏了很多现有的 2.*x* 版代码，一些代码比以前运行得更慢。尽管会有这样的问题，我们还是建议所有人都尽快升级到 Python 3（Python 2.*x* 的维护期到 2020 年就结束了），因为随着 3.*x* 系列的逐渐成熟，大多数问题已经被解决。实际上，本书中的很多新语言特性就来自于 Python 3。

本书使用的是 Python 3.6。

如果想要阅读更多相关内容，可以查看 Ed Schofield 在 Python-Future 网站上提供的资源，以及 Nick Coghlan 关于这种版本转换的详细指南。

SciPy生态系统和社区

SciPy 是一个功能丰富的主流库。与 NumPy 一样，它也是 Python 的杀手级应用之一。在 SciPy 功能的基础之上，衍生了大量相关库，本书中涉及了其中很多库。

这些库的作者和用户在世界各地的很多活动和会议上聚集，包括一年一度的美国奥斯汀 SciPy 大会、EuroSciPy、SciPy India、PyData，等等。我们强烈建议你参加其中一项活动，与 Python 世界中最优秀的科学软件的开发者会面。如果你无法亲自前往，或者只想感受一下这些会议的气氛，很多人会在网络上发布他们的演讲，你可以看一下。

免费的开源软件

SciPy 社区支持开源软件开发，几乎所有 SciPy 库的源代码都可以免费获取，任何人都可以阅读、编辑和重用这些源代码。

如果希望别人使用你的代码，最好的一种实现方式就是让代码免费而且公开。如果使用了封闭源码的软件，但它不能完全满足你的需求，那只能说你运气不佳。你可以给开发者发邮件，请求他们添加新的功能（通常无济于事），也可以自己开发新的软件。但如果代码是开源的，那么你就可以轻松地使用从本书中学到的技能来添加或修改软件的功能。

同样，如果你发现了一个软件的 bug，那么对用户和开发者来说，能够接触到源代码可以使事情变得更加容易。即使不能完全理解代码，通常也可以更加深入地诊断问题，并帮助开发者进行修复，这对所有人来说都是很好的学习经历。

开放的源码，开放的科学

在科学软件开发中，上述场景都极其常见，而且非常重要：科学软件一般都是建立在前人的工作基础之上，或者是对其做了一些有趣的修改。此外，因为科学成果发布和改进的节奏非常快，很多代码在发布之前都没有经过充分的测试，所以科学软件或多或少都有 bug。

代码开源的另一个重要原因是为了促进可重复性研究。许多人都有过这种经历：读了一篇非常精彩的论文，然后下载代码并在自己的数据上进行试验，结果却发现可执行文件不是为自己的操作系统编译的，程序无法运行，程序有 bug，某个功能缺失，或者产生了出人意料的结果。通过将科学软件开源，不但可以提高软件质量，还可以使科学发现过程一目了然，比如假设是如何做出的，哪些地方有硬编码？开源可以解决很多类似的问题。开源还允许科研人员基于同行的代码继续开发，培养新的协作关系，加速科学研究的进程。

开放源码许可证

如果想让别人使用你的代码，你**必须**选择一种开源许可证。如果没有开源许可证，那么代码默认就是封闭的。即使你公开了代码（比如将代码放在一个公开的 GitHub 仓库中），没有软件许可证的话，任何人也都不能使用、修改或重新发布你的代码。

在众多许可证选项中进行选择时，你必须先确定允许别人用你的代码做什么。你想让别人通过销售你的代码或使用了你的代码的软件来获取利润吗？你想限制代码仅供免费软件使用吗？

免费的开源软件许可证有两大类：

- 宽松式（permissive）许可证
- copy-left 许可证

宽松式许可证表明你授予任何人以任何方式使用、编辑和重新发布你的代码的权利，包括将你的代码作为商业软件的一部分。常见的这类许可证包括 MIT 许可证和 BSD 许可证。SciPy 社区采用的是新 BSD（又称“修正 BSD”或“三条款 BSD”）许可证。使用这种许可证意味着接受各种人员贡献的代码，包括工业界和创业公司的代码。

copy-left 许可证也允许他人使用、编辑和重新发布你的代码。但是，这种许可证还规定衍生代码必须用 copy-left 许可证发布。通过这种方式，copy-left 许可证对代码的用途进行了限制。

最常见的 copy-left 许可证是 GPL（GNU Public License）。使用 copy-left 许可证的最大问题是，经常会使那些来自于私人部门的潜在用户和贡献者无法使用你的代码，甚至包括未来的你！这会严重削减你的用户基础，进而影响软件取得的成就。在科学界，这就意味着更少的引用数。

如果想在许可证的选择方面获得更多帮助，可以看一下 Choose a License 这个网站。如果从科研角度考虑，我们推荐你看一下 Jake VanderPlas 的博文“The Whys and Hows of Licensing Scientific Code”，他是华盛顿大学自然科学研究主任，不折不扣的 SciPy 超级明星。实际上，Jake 的这段话非常清楚地解释了软件许可制度：

> ……如果你想从这篇文章中总结出 3 条信息，那么就是以下 3 条。
>
> (1) 一定要对代码进行许可。未许可的代码是封闭代码，因此任何开放许可都优于没有许可（见第 2 条）。
>
> (2) 一定要使用 GPL 兼容的许可证。GPL 兼容许可证可以确保你的代码具有广泛的兼容性，这种许可证包括 GPL、新 BSD 以及其他一些许可证（见第 3 条）。
>
> (3) 一定要使用宽松式的、BSD 风格的许可证。相比于 GPL 或 LGPL 这样的 copy-left 许可证，更应该使用新 BSD 或 MIT 这样的宽松式许可证。

本书中的所有代码都具有三条款 BSD 许可证。其中包括来自于他人的源代码片段，这些代码通常具有某种宽松式开放许可（尽管不一定是 BSD）。

对于你自己的代码，我们建议你遵循社区的做法。例如，Python 科学应用中使用的是三条款 BSD 许可证，但 R 语言社区采用的则是 GPL 许可证。

GitHub：实现社交化编码

前面讨论过了使用开源许可证发布源代码，这样做很可能会促进大量人员下载并使用你的代码，修改其 bug 并添加新的功能。你应该将代码放在何处，以便人们找到它呢？ bug 修改和新功能如何补充到代码中呢？如何跟踪这些问题和变更呢？可以想象，如果没有有效的管理手段，这些问题很快就会失控。

答案就是：GitHub。

GitHub 是一个用于托管、分享和开发代码的网站，它是建立在版本控制软件 Git 基础之上的。一些非常出色的资源可以供你学习如何使用 GitHub，比如 Peter Bell 和 Brent Beer 合著的 *Introducing GitHub*。因为 SciPy 生态系统中的绝大多数项目都托管在 GitHub 上，所以学习一下它的使用方法是很有必要的。

GitHub 对开源代码的贡献起到了巨大的推进作用，它允许用户发布代码并自由协作。任何人都可以复制一份代码（称为 fork）并修改其核心内容，然后创建一个 pull request 将修改贡献给原来的代码。GitHub 中有一些非常贴心的特性，比如管理问题和变更请求，以及确定谁可以直接修改你的代码。你甚至可以跟踪修改、贡献者和其他有趣的统计信息。GitHub 还有很多其他精彩特性，但我们要将大部分留给你自己去发现，本书后面会涉及其中一些特性。从本质上讲，GitHub 使软件开发变得更加大众化（见图 P-1），大大降低了入门的门槛。

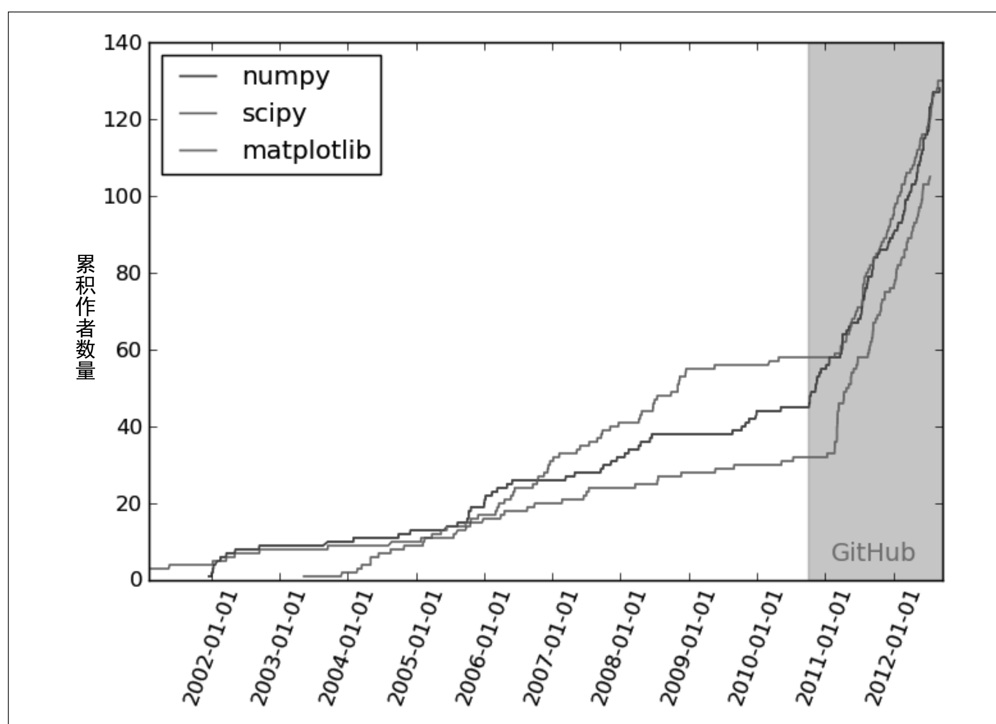


图 P-1：GitHub 的影响（经作者 Jake VanderPlas 授权使用）

为SciPy生态系统贡献你的力量

如果积累了更多 SciPy 经验并开始用它进行科学研究，你可能会发现某个包缺少你需要的功能，也可能认为自己的程序更高效，还可能发现某个程序的 bug。如果你达到了这个程度，那么就可以开始为 SciPy 生态系统贡献力量了。

我们强烈建议你做这件事。社区能够生存发展，就是因为人们乐于分享自己的代码和改进

现有的代码。聚沙成塔，集腋成裘。除去无私的贡献精神外，贡献代码还能获得一些非常实际的个人利益。积极地参与社区可以让你成为一名更加优秀的程序员，你贡献的任何代码都会被其他人审查和反馈意见。你还可以学会如何使用 Git 和 GitHub，它们是维护和分享代码的非常有用的工具。你甚至还会发现，与 SciPy 社区互动会为你构建一个更加广阔的科研网络，并提供意想不到的工作机会。

我们希望你不只是一位 SciPy 用户。你加入的是一个社区，你的工作会为它锦上添花，使所有从事科学编程的人员受益匪浅。

Python中的一些恶搞

如果你担心 SciPy 社区对新手来说是一个充满压力的地方，那大可不必，因为这个社区是由与你非常相似的人员组成的，都是一些科研工作者，他们通常具有很强的幽默感。

在 Python 世界中，你肯定会发现一些与巨蟒喜剧团（Monty Python）有关的梗。Airspeed Velocity[4] 是一个测量软件速度的包（后面会做更多介绍），其中就引用了《巨蟒与圣杯》中的一句台词："一只没有衔任何东西的燕子的飞行速度是多少？"

另一个有着搞笑名称的包是 Sux，它允许你在 Python 3 环境中使用 Python 2 的包。这是使用新西兰口音对 six 包的恶搞，这个包允许你在 Python 2 中使用 Python 3 的语法。转换到 Python 3 后，Sux 语法可以让你在使用那些仅用于 Python 2 的包时不那么沮丧。

```
import sux
p = sux.to_use('my_py2_package')
```

一般来说，Python 库的名字都起得十分绚丽，希望你能想出一些这样的名字！

获取帮助

当遇到困难时，我们做的第一件事就是到网上搜索自己要完成的任务或收到的出错信息。这就要用到 Stack Overflow，一个关于编程的非常出色的问答网站。如果没有立刻找到所需的内容，可以试着扩充一下搜索条件，找到与你有同样问题的人。

有时你可能是第一个遇到这种问题的人（尤其是使用一个新包时），这时也不要惊慌失措！正如前面所说，SciPy 社区用户是一帮非常友好的家伙，他们分布在互联网的各个地方。接下来你应该做的就是搜索 `<library name> mailing list`，找到相应的邮件列表来寻求帮助。库的作者和高级用户经常阅读邮件列表，而且对新人非常热情。注意，需要先**订阅**邮件列表，然后才能发邮件，这是一种基本礼仪。否则，在允许你的邮件出现在邮件列表前，经常需要有人手动检查它是否为一封垃圾邮件。有时加入一个邮件列表会令人厌烦，但我们强烈推荐你使用它，它绝对是一个学习的好地方！

安装Python

本书假设你安装了 Python 3.6（或更新的版本）和本书需要的所有 SciPy 包。在随书数据

注 4：中文意思是"飞行速度"。——编者注

中，我们还包括了一个 environment.yml 文件，并在其中列出了所有需要的包及其版本。获得全部所需包的最简单方式是先安装 conda，它是一个管理 Python 环境的工具，然后再将 environment.yml 文件传给 conda，以便一次性安装所有包的正确版本。

```
conda env create --name elegant-scipy -f path/to/environment.yml
source activate elegant-scipy
```

访问本书的 GitHub 仓库以获取更多细节。

获取随书资料

本书的所有代码和数据都可以在 GitHub 仓库（https://github.com/elegant-scipy/elegant-scipy）中找到。在该仓库的 README 文件中，你会发现根据 markdown 源文件建立 Jupyter 笔记本的相关指导。建立了 Jupyter 笔记本之后，你就可以使用仓库中的数据交互式地运行它们。

我们开始吧

本书综合了 SciPy 社区提供的一些最优雅的代码。在学习的过程中，我们还会介绍 SciPy 社区解决的一些实际科学问题。本书还会带你了解一个期待你加入其中的热情的、协作式的科学编程社区的面貌。

欢迎阅读本书。

排版约定

本书将使用如下排版约定。

- **黑体**
  表示新术语或重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键词等。
- 加粗等宽粗体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用颜色

本书中的一些示例使用了不同的颜色，但纸质书中是看不出颜色的。你可以查看 https://github.com/elegant-scipy/elegant-scipy 中的原版电子书。

使用代码示例

本书的附加资料（示例代码、练习题等）可以从 https://github.com/elegant-scipy/elegant-scipy 下载。[5]

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如，"*Elegant SciPy* by Juan Nunez-Iglesias, Stéfan van der Walt, and Harriet Dashnow (O'Reilly). Copyright 2017 Juan Nunez-Iglesias, Stéfan van der Walt, and Harriet Dashnow, 978-1-491-92287-3"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari



Safari（原来叫 Safari Books Online）是一个会员制的培训和参考平台，面向企业、政府、教育从业者和个人。

会员可以访问几千种图书、培训视频、学习路径、互动式教程和精选播放列表，提供这些资源的出版商超过 250 家，包括 O'Reilly Media、Harvard Business Review、Prentice Hall

注 5：也可以访问本书图灵社区页面（http://www.ituring.com.cn/book/2078），下载本书示例代码及彩图，并提交中文版勘误。——编者注

Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

要获得更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我们必须向为本书做出重要贡献的众多人员致以诚挚的谢意，没有他们的帮助，我们根本不可能完成本书。

首先，感谢 NumPy、SciPy 及其相关库的众多贡献者，希望本书能真正体现出你们无与伦比的工作。

其次，感谢更广泛的 Python 科学生态系统中的贡献者，包括为本书若干章提供程序基础的人：Vighnesh Birodkar、Matt Rocklin 和 Warren Weckesser。由于出版时间的关系，有些贡献者提供的内容没有出现在本书中，我们也必须向你们表示感谢，你们的工作启发了我们，希望本书未来的版本中能够包含你们的工作。我们还要感谢 Nicolas Rougier 提供了很多建议，我们将这些建议放在了示例和练习中。

还有很多人提供了数据和代码，为我们节省了大量搜索和探查的时间。感谢 Lav Varshney 提供的用于蠕虫大脑光谱图布局的 MATLAB 源代码（第 3 章和第 6 章），以及 Stefano

Allesina 提供的 St. Marks 食品网站数据（第 6 章）。

非常感谢在本书出版前做出更正和提出建议的人们，包括 Bill Katz、Matthias Bussonnier 和 Mark Hyun-ki Kim。

还要感谢我们的技术审阅人，Thomas Caswell、Nelle Varoquaux、Lav Varshney 和 Greg Wilson，他们从繁忙的工作中挤出时间为我们梳理草稿，并无私地分享了他们的专业知识。

尽管我们会根据读者的意见继续修订和改进本书，但一些朋友和家人对本书的早期版本进行了校对，并提供了宝贵的意见、建议和鼓励，这让我们受益良多。Malcolm Gorman、Alicia Oshack、PW van der Walt、Simon Kocbek、Nelle Varoquaux 和 Ariel Rokem，谢谢你们。

当然，还要感谢 O'Reilly 出版社的编辑 Meg Blanchette、Brian MacDonald 和 Nan Barber。我们要特别感谢 Meg，她最先接触我们，洽谈本书的出版事宜，并在我们还一头雾水的时候给出了非常有价值的早期指导意见。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章

优雅的NumPy：Python科学应用的基础

> （NumPy）无处不在，遍布我们的周围。它甚至现在就在这间屋子里。不论望向窗外，还是打开电视机，你都能看见它。你去上班、去教堂，甚至去缴税时，都能感觉到它。
>
> ——墨菲斯，《黑客帝国》

本章将介绍 SciPy 中的几个统计函数，此外还将重点介绍 NumPy 数组，这种数据结构是 Python 中几乎所有科学数值计算的基础。我们将看到 NumPy 数组操作如何用简洁而又高效的代码来处理数值型数据。

我们的用例是利用基因表达数据预测皮肤癌患者的死亡率，这些数据来自癌症和肿瘤基因图谱（TCGA，the cancer genome atlas）计划。本章和第 2 章的全部工作都是为了实现这一目标，同时还要学习 SciPy 中的一些关键概念。在预测死亡率之前，我们需要使用一种称为 RPKM 标准化的方法将基因表达数据标准化，这样可使不同样本和基因的测量结果具有可比性。（稍后将解释"基因表达"这个词的意义。）

我们先用一段代码来热身，并介绍本章的主旨。每一章都会从一段示例代码开始，我们确信这段代码能够体现 SciPy 生态系统中某个函数的优雅和强大。本章示例将重点介绍 NumPy 的向量化和广播规则，它们可以使我们非常高效地操纵和推理数据数组。

```
def rpkm(counts, lengths):
    """Calculate reads per kilobase transcript per million reads.

    RPKM = (10^9 * C) / (N * L)
```

```
    Where:
    C = Number of reads mapped to a gene
    N = Total mapped reads in the experiment
    L = Exon length in base pairs for a gene

    Parameters
    ----------
    counts: array, shape (N_genes, N_samples)
        RNAseq (or similar) count data where columns are individual samples
        and rows are genes.
    lengths: array, shape (N_genes,)
        Gene lengths in base pairs in the same order
        as the rows in counts.

    Returns
    -------
    normed : array, shape (N_genes, N_samples)
        The RPKM normalized counts matrix.
    """
    N = np.sum(counts, axis=0) # 将每一列相加以得到每个样本的总read数
    L = lengths
    C = counts

    normed = 1e9 * C / (N[np.newaxis, :] * L[:, np.newaxis])

    return(normed)
```

这个示例演示了 NumPy 数组使代码更加优雅的几种方法。

- 数组可以是一维的（如列表），可以是二维的（如矩阵），也可以是更高维度的，这使得数组可以表示多种形式的数值型数据。示例中操作的是二维矩阵。
- 数组可以沿着**轴**进行操作。在第一行代码中，通过指定 `axis=0` 沿着每一列进行加总。
- 数组可以一次性进行多个数值操作。例如，在函数末尾，我们用保存计数的二维数组（`C`）除以保存列总和的一维数组（`N`）。这就是广播机制，稍后会对其原理进行详细介绍。

在深入研究 NumPy 的强大功能之前，我们先花点时间介绍一下要处理的生物学数据。

1.1 数据简介：什么是基因表达

我们将完成一项**基因表达分析**，以演示 NumPy 和 SciPy 解决实际生物学问题的强大能力。我们将使用建立在 NumPy 之上的 pandas 库来读取和整理数据文件，然后在 NumPy 数组中高效地处理数据。

根据**分子生物学中心法则**，运行一个细胞（有机体同样如此）所需的所有信息都存储在一个称为**脱氧核糖核酸**（deoxyribonucleic acid）的分子中，又称 DNA。这种分子有一个重复性骨架，上面顺次分布着一种称为碱基（base）的化学成分（见图 1-1）。碱基有四种类型，缩写分别是 A、C、G 和 T，它们构成了保存生物信息的基本结构。

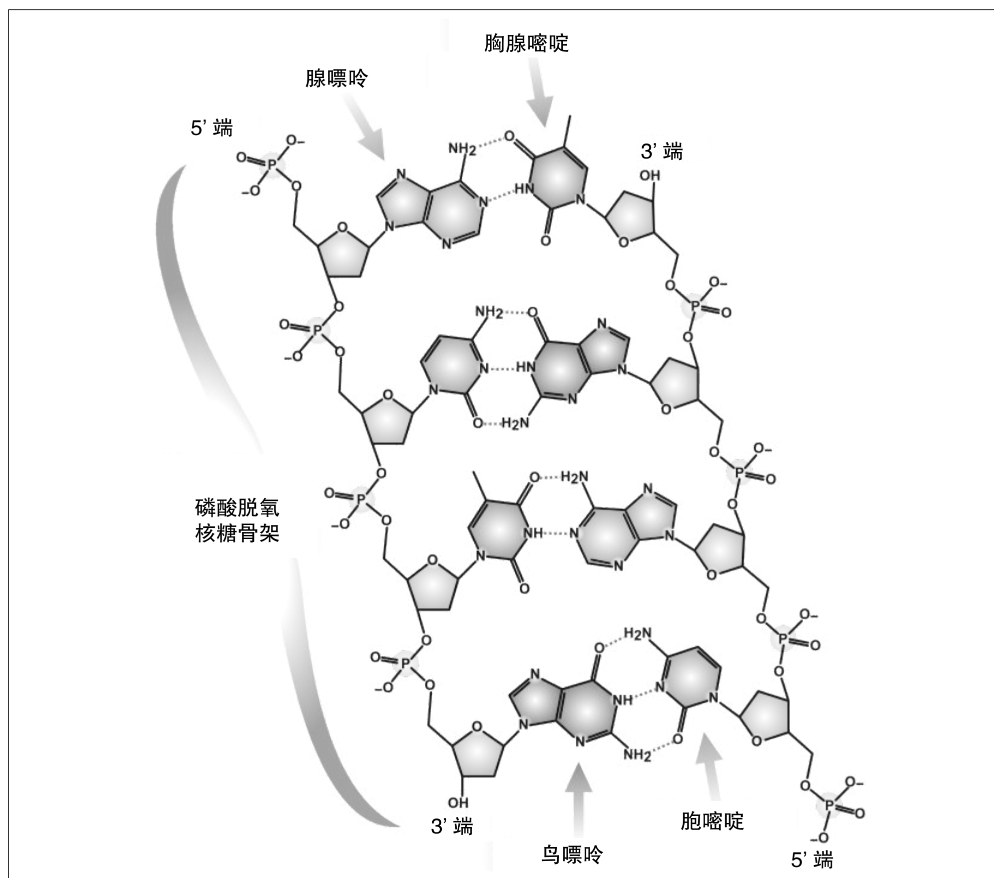


图 1-1：DNA 的化学结构（图的作者为 Madeleine Price Ball，根据 CC0 公有领域许可条款使用）

为获取这种信息，DNA 被**转录**为一种姐妹分子，称为**信使核糖核酸**（mRNA，messenger ribonucleic acid）。最后，这种 mRNA 被**翻译**为蛋白质，它是构成细胞的主要物质（见图 1-2）。编码信息（经由 mRNA）以制造蛋白质的 DNA 片段称为基因。

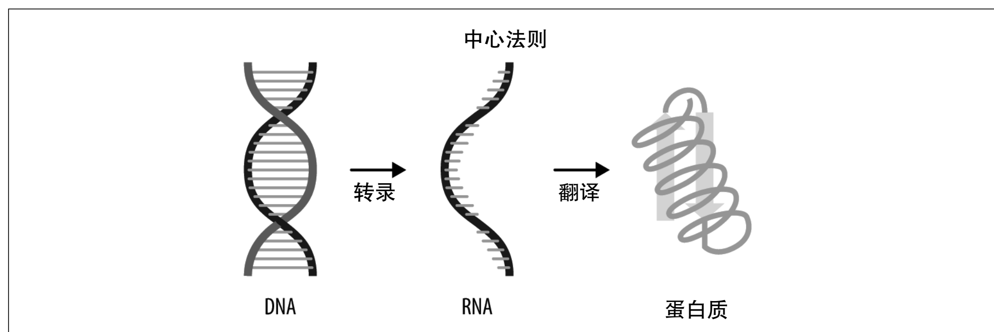


图 1-2：分子生物学中心法则

由某种基因生成的 mRNA 数量称为这种基因的**表达**。尽管理想的做法是测量蛋白质的表达水平，但这比测量 mRNA 要困难得多。好在 mRNA 的表达水平通常与相应的蛋白质表达水平是相关的。[1] 因此，通常测量 mRNA 的表达水平并在此基础上进行分析。正如你将在后面看到的，这一般没有什么问题，因为我们使用 mRNA 水平的目的是预测生物学结果，而不是对蛋白质进行明确的说明。

需要注意的是，你体内所有细胞中的 DNA 是完全相同的。因此，细胞间的差异来自从 DNA 转录为 RNA 时的**差异性表达**：在不同的细胞中，DNA 的不同部分会加工处理成下游分子（见图 1-3）。类似地，我们将在本章及下一章中看到，差异性表达可以区分出不同类型的癌症。

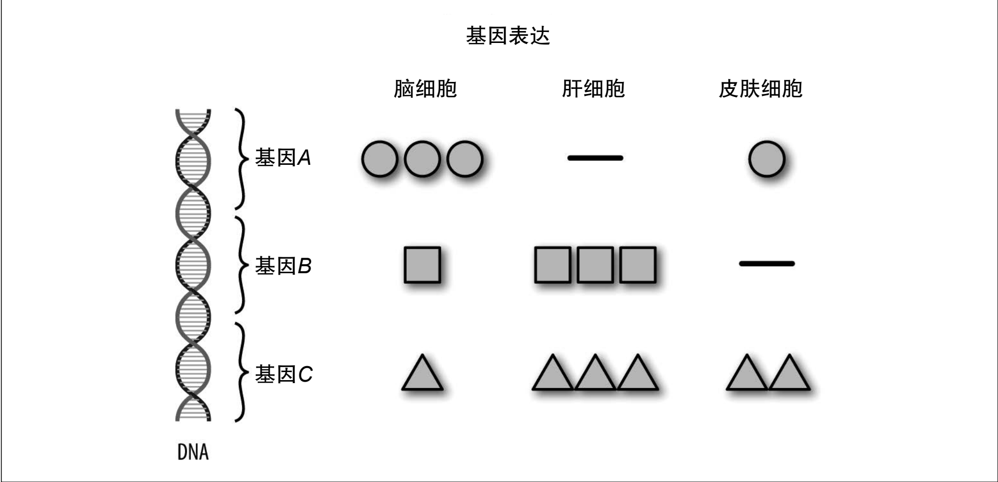


图 1-3：基因表达

当前最先进的 mRNA 测量技术称为 RNA 测序（RNAseq）。先从一个组织样本（如患者的活体组织检查样本）中提取出 RNA，通过**反转录**将其转换为（更加稳定的）DNA，然后读取出那些在组装 DNA 序列时能发出荧光的经过化学修饰的碱基。目前，高通量测序仪器只能读取很短的片段（通常在 100 个碱基左右），这种 DNA 短序列就称为 read。我们要测量数百万个 read，然后根据它们的顺序计算出来自每个基因的 read 数量（见图 1-4）。我们将直接使用这些计数数据开始分析。

注 1：MAIER T, GÜELL M, SERRANO L. Correlation of mRNA and protein in complex biological samples [J]. FEBS Letters, 2009, 583(24): 3966-3973.

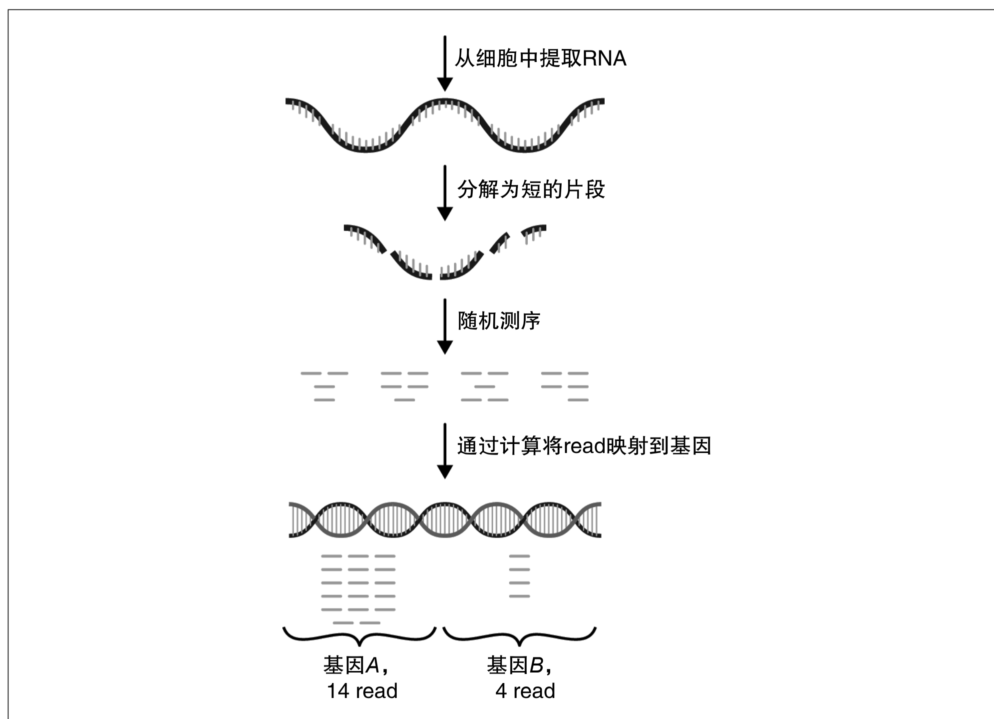


图 1-4：RNA 测序（RNAseq）

表 1-1 展示了基因表达计数数据的一个极小样本。

表1-1　基因表达计数数据

	细胞类型*A*	细胞类型*B*
基因 0	100	200
基因 1	50	0
基因 2	350	100

这份数据是一张计数表格，其中的整数表示在每种细胞类型中对每种基因观察到的 read 数量。看到不同细胞类型的每种基因在计数上的差别了吗？我们可以用这样的信息来找出两种细胞间的差别。

在 Python 中表示这种数据的一种方法是使用列表的列表。

```
gene0 = [100, 200]
gene1 = [50, 0]
gene2 = [350, 100]
expression_data = [gene0, gene1, gene2]
```

在以上代码中，每种基因在不同细胞类型上的表达被保存在一个 Python 整型列表中。然后我们将这些列表保存在一个列表（如果原意，你可以称其为**元列表**，meta-list）中。可以用两级列表索引提取出单个数据点。

```
expression_data[2][0]
```

```
350
```

鉴于 Python 解释器的工作方式，这样保存数据点是效率非常低的一种方法。首先，Python 列表都是**对象**的列表，因此上面的 `gene2` 列表不是整数列表，而是一个指向整数的**指针**列表，这会带来不必要的开销。其次，这种方式意味着将列表和整数随机地保存在计算机 RAM 中完全不同的区域。但是现代处理器更喜欢按**块**读取内存中的内容，因此，将数据分散保存在 RAM 中是非常低效的。

这正是 NumPy 数组要解决的问题。

1.2　NumPy的*N*维数组

NumPy 的一种主要数据类型是 *N* 维数组（`ndarray`，简称数组）。*N* 维数组是 SciPy 中很多高级数据处理技术的基础。本节将详细介绍向量化和广播技术，利用它们可以写出强大而又优雅的代码来处理数据。

首先研究一下 *N* 维数组。这些数组必须是同质的：数组中的所有项都必须是同一类型。在上面的示例中，我们需要保存整数。之所以称为 *N* 维数组，是因为它可以有任意数量的维度。一维数组基本上等价于 Python 列表。

```
import numpy as np

array1d = np.array([1, 2, 3, 4])
print(array1d)
print(type(array1d))

[1 2 3 4]
<class 'numpy.ndarray'>
```

数组具有特殊的属性和方法，在数组名称后面加一个点后就可以使用这些属性和方法了。例如，可以使用以下代码得到数组的**形状**。

```
print(array1d.shape)
```

```
(4,)
```

结果是只有一个数值的元组。或许你想知道：为什么不像对待列表那样使用 `len` 方法？这里确实可以使用 `len`，但它不能扩展到二维数组。

表 1-1 中数据的表示方法如下所示。

```
array2d = np.array(expression_data)
print(array2d)
print(array2d.shape)
print(type(array2d))

[[100 200]
 [ 50   0]
 [350 100]]
```

```
(3, 2)
<class 'numpy.ndarray'>
```

由上可见，`shape` 属性扩展了 `len`，以表示出数组中多个维度上的数据大小（见图 1-5）。

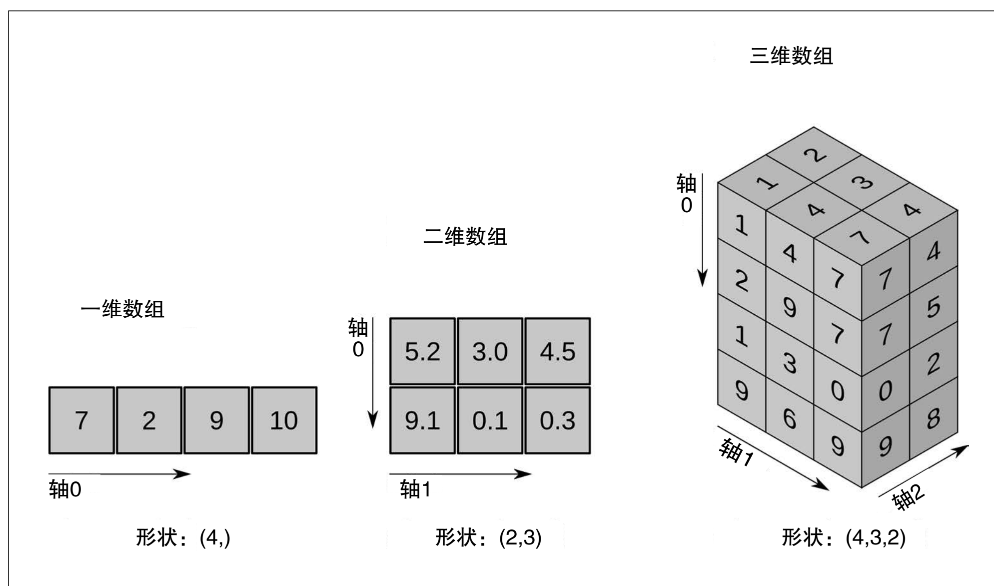


图 1-5：NumPy 的 *N* 维数组在一维、二维和三维上的可视化表示

数组还有其他属性，比如表示维度数量的 `ndim`。

```
print(array2d.ndim)
```

```
2
```

当你用 NumPy 完成自己的数据分析任务时，你就会逐渐熟悉这些属性和方法。

NumPy 数组可以表示具有更多维度的数据，比如核磁共振成像（MRI，magnetic resonance imaging）数据，其中包括对三维立体数据的测量。如果想要保留随时间变化的 MRI 数据，那么就需要四维 NumPy 数组。

本章主要探讨二维数据，后面将介绍高维数据，并教你如何编写代码来处理任意维度的数据。

1.2.1 为什么用*N*维数组代替Python列表

数组的速度非常快，因为它支持向量化操作。向量化操作由低级语言 C 编写而成，可以作用于整个数组。如果你有一个列表，而且想将列表中的每个元素都乘以 5，那么标准的 Python 实现方式就是编写一个循环语句，在列表的元素之间迭代，将每个元素都乘以 5。然而，如果数据是用数组表示的，那么你就可以一次性将数组中的所有元素都乘以 5。高度优化的 NumPy 库会在后台尽快完成这些迭代。

```
import numpy as np

# 创建一个取值范围为0~1 000 000（不含）的N维整数数组
array = np.arange(1e6)

# 将数组转换成列表
list_array = array.tolist()
```

我们用 IPython 中的 `timeit` 函数来比较一下将数组中所有值乘以 5 所用的时间。先看看数据在列表中的情况。

```
%timeit -n10 y = [val * 5 for val in list_array]

10 loops, average of 7: 102 ms +- 8.77 ms per loop (using standard deviation)
```

然后使用 NumPy 中内置的**向量化**操作。

```
%timeit -n10 x = array * 5

10 loops, average of 7: 1.28 ms +- 206 µs per loop (using standard deviation)
```

比原来快了 50 多倍，而且代码更简洁！

数组还比列表具有更高的存储效率。在 Python 中，列表中的每个元素都是一个对象，并进行了健康的内存分配。（真的健康吗？）相比之下，数组中的每个元素只占用必要的内存。例如，在一个 64 位的整数数组中，每个元素占用的空间就是 64 位；除此之外，数组还有一些微不足道的额外开销，用于存储元数据，比如前面讨论过的 `shape` 属性。这种存储方式占用的空间通常远远小于 Python 列表中的对象所占用的空间。（如果想要更加深入地研究一下 Python 的内存分配原理，可以阅读 Jake VanderPlass 的博文“Why Python Is Slow: Looking Under the Hood”。）

此外，当用数组进行计算时，你还可以使用**切片**操作**在不复制基础数据的情况下**取数组的子集。

```
# 创建一个N维数组x
x = np.array([1, 2, 3], np.int32)
print(x)

[1 2 3]

# 创建x的一个“切片”
y = x[:2]
print(y)

[1 2]

# 将y的第一个元素设置为6
y[0] = 6
print(y)

[6 2]
```

注意，尽管我们编辑了 `y`，但 `x` 也被修改了，因为 `y` 和 `x` 引用的是同一数据！

```
# 现在x中的第一个元素变成6了
print(x)
```

```
[6 2 3]
```

这意味着进行数组引用操作时一定要小心。如果想在处理数据的同时不改变初始数据，就应该复制数据。

```
y = np.copy(x[:2])
```

1.2.2 向量化

前面讨论过数组操作的速度。NumPy 用来提高数组操作速度的一项诀窍就是**向量化**。向量化无须使用 for 循环就可以对数组中的每个元素进行计算。除了能提高数组操作的速度，它还可以使代码更自然易读。下面来看几个示例。

```
x = np.array([1, 2, 3, 4])
print(x * 2)
```

```
[2 4 6 8]
```

这里 x 数组中有 4 个值，我们隐式地将 x 中的每个元素都乘以单一值 2。

```
y = np.array([0, 1, 2, 1])
print(x + y)
```

```
[1 3 5 5]
```

我们将 x 中的每个元素与 y 中的对应元素相加，y 是与 x 形状相同的数组。

我们希望这两个操作简单而又直观地说明了什么是向量化。NumPy 使得向量化的速度非常快，比手动迭代数组要快得多。（你可以使用 IPython 中的命令 %timeit 验证一下，做法前面提过。）

1.2.3 广播

广播是在两个数组间执行隐式操作的一种方法，它是 N 维数组中最强大却经常被误解的功能之一。广播允许你在形状**兼容**的两个数组间执行操作，它可以创建出比任何一个初始数组都大的数组。例如，通过恰当地重塑两个向量，可以计算出它们的外积。

```
x = np.array([1, 2, 3, 4])
x = np.reshape(x, (len(x), 1))
print(x)
[[1]
 [2]
 [3]
 [4]]

y = np.array([0, 1, 2, 1])
y = np.reshape(y, (1, len(y)))
print(y)

[[0 1 2 1]]
```

对于两个数组的每个维度，如果其中一个等于 1，或者两个维度彼此匹配，那么就可以说这两个数组的形状是兼容的。[2]

我们检查一下这两个数组的形状。

```
print(x.shape)
print(y.shape)

(4, 1)
(1, 4)
```

两个数组都有两个维度，而且内侧维度都等于 1，因此这两个数组的形状是兼容的！

```
outer = x * y
print(outer)

[[0 1 2 1]
 [0 2 4 2]
 [0 3 6 3]
 [0 4 8 4]]
```

外侧维度可以告诉我们结果数组的大小。在这个示例中，我们可以得到一个形状为 `(4, 4)` 的数组。

```
print(outer.shape)

(4, 4)
```

你可以检查一下，对于所有的 `(i, j)`，都有 `outer[i, j] = x[i] * y[j]`。

这种操作是根据 NumPy 的**广播法则**完成的，它隐式地扩展了一个数组中长度为 1 的维度，以匹配另一个数组中相应的维度。稍安勿躁，本章后面会更加详细地讨论这些规则。

我们将在本章余下的内容中看到，当探索实际数据时，广播机制对于数组数据的计算极其有价值，它使我们可以简洁而又高效地实现非常复杂的操作。

1.3 探索基因表达数据集

我们将要使用的数据集是一份皮肤癌样本的 RNA 测序实验数据，它来自 TCGA 计划。我们已经对数据进行了清洗和排序，你可以直接使用本书仓库中的 data/counts.txt 文件。

在第 2 章中，我们将使用这份基因表达数据来预测皮肤癌患者的死亡率，并为 TCGA 组织的一篇论文中的图 5A 和图 5B 重新制作一份简化的版本。但我们先要搞清楚数据中的偏差，并思考如何改进。

用pandas读取数据

首先，用 pandas 在计数表格中读取数据。pandas 是一个专门用于数据处理和分析的库，

注 2：我们总是从最后一个维度开始比较，并逐步向前推进。如果一个数组的维度数量比另一个多，则忽略多余的维度。例如，`(3, 5, 1)` 和 `(5, 8)` 是可以匹配的。

它特别重视对表格数据和时间序列数据的处理。这里我们将用它读取混合类型的表格数据。pandas 使用的是 `DataFrame` 类型，这是一种非常灵活的表格形式，基于 R 语言中的数据框对象开发而成。例如，我们将读取的数据中有一列是基因名称（字符串），还有多列是计数数据（整数），因此，将其读取到具有同质数据的数组中是一种错误的做法。虽然 NumPy 对于混合数据类型（称为“结构化数组”）有一定的支持，但其设计初衷并不是处理这种情况，这会使得随后的操作产生一些不必要的问题。

将数据读取到 pandas 数据框中可以让 pandas 完成所有解析工作，然后提取出相关信息并保存到更高效的数据类型中。本章主要用 pandas 完成数据导入，后面的章节会使用 pandas 的更多功能。如果想要详细学习 pandas，可以阅读 pandas 创建者 Wes McKinney 的著作《利用 Python 进行数据分析》。

```
import numpy as np
import pandas as pd

# 导入TCGA黑色素瘤数据
filename = 'data/counts.txt'
with open(filename, 'rt') as f:
    data_table = pd.read_csv(f, index_col=0) # 用pandas解析文件

print(data_table.iloc[:5, :5])
```

```
       00624286-41dd-476f-a63b-d2a5f484bb45  TCGA-FS-A1Z0  TCGA-D9-A3Z1  \
A1BG                                1272.36        452.96        288.06
A1CF                                   0.00          0.00          0.00
A2BP1                                  0.00          0.00          0.00
A2LD1                                164.38        552.43        201.83
A2ML1                                 27.00          0.00          0.00
       02c76d24-f1d2-4029-95b4-8be3bda8fdbe  TCGA-EB-A51B
A1BG                                 400.11        420.46
A1CF                                   1.00          0.00
A2BP1                                  0.00          1.00
A2LD1                                165.12         95.75
A2ML1                                  0.00          8.00
```

可以看出，pandas 贴心地提取出了标题行，并用它对各列进行了命名。第一列给出了每种基因的名称，其余各列表示独立的样本。

我们还需要一些相应的元数据，其中包括样本信息和基因长度。

```
# 样本名称
samples = list(data_table.columns)
```

我们还需要一些关于基因长度的信息，以便进行标准化。为了能够使用 pandas 索引的一些奇妙功能，我们将 pandas 表中的第一列基因名称设定为索引。

```
# 导入基因长度
filename = 'data/genes.csv'
with open(filename, 'rt') as f:
    # 使用pandas解析文件，以GeneSymbol作为索引
    gene_info = pd.read_csv(f, index_col=0)
```

```
print(gene_info.iloc[:5, :])

            GeneID GeneLength
GeneSymbol
CPA1          1357       1724
GUCY2D        3000       3623
UBC           7316       2687
C11orf95     65998       5581
ANKMY2       57037       2611
```

我们检查一下基因长度数据与计数数据的匹配情况。

```
print("Genes in data_table: ", data_table.shape[0])
print("Genes in gene_info: ", gene_info.shape[0])

Genes in data_table: 20500
Genes in gene_info: 20503
```

与实验中实际测量的基因相比，基因长度数据中的基因数量更多。我们需要筛选基因长度数据，只保留那些相关的基因，并确保它们与计数数据中的基因具有相同的顺序。这就到了 pandas 索引大显身手的时候了！我们可以找出两种源数据中基因名称的交集，然后用它来索引两个数据集，确保两个数据集具有同样的基因和顺序。

```
# 取基因信息的子集来匹配计数数据
matched_index = pd.Index.intersection(data_table.index, gene_info.index)
```

现在，用基因名称的交集来索引计数数据。

```
# 二维数组包含了每个独立样本中每种基因的表达计数
counts = np.asarray(data_table.loc[matched_index], dtype=int)

gene_names = np.array(matched_index)

# 检查测量的基因数和独立样本数
print(f'{counts.shape[0]} genes measured in {counts.shape[1]} individuals.')

20500 genes measured in 375 individuals.
```

还有基因长度数据：

```
# 一维数组包含了每种基因的长度
gene_lengths = np.asarray(gene_info.loc[matched_index]['GeneLength'],
                          dtype=int)
```

再检查一下对象的维度：

```
print(counts.shape)
print(gene_lengths.shape)

(20500, 375)
(20500,)
```

不出所料，它们匹配得非常完美！

1.4 标准化

真实世界中的数据包含了各种各样的测量方法，在使用它们进行任意类型的分析前，对其进行检查，以确定是否需要标准化，是非常重要的。例如，使用数字温度计进行测量与使用水银温度计并由人读数之间有系统性的差别。因此，做样本比较时经常要做一定的数据整理工作，让所有测量结果具有同样的尺度。

就我们的示例来说，需要确保揭示出的任何数据差异都是由真实的生物学差异造成的，而不是由测量的技术手段造成的。我们将考虑两种层次的标准化，它们经常联合应用于基因表达数据集，这就是样本（列）间的标准化和基因（行）间的标准化。

1.4.1 样本间的标准化

例如，在 RNA 测序实验中，独立样本间的计数值会相差很大。我们看一下基因表达计数数据在所有基因间的分布。首先，对列进行加总，以得到每个独立样本中所有基因表达的总计数值，这样就可以检查独立样本间的变动了。为了对总计数分布进行可视化，我们将使用核密度估计（KDE，kernel density estimation），这是一种常用于对直方图进行平滑的技术，它可以更加清晰地描绘出基础分布。

在开始之前，需要先进行一些绘图设置工作（每一章都要这样做）。请看下方附注栏“有关绘图的简单介绍”，以了解下面每行代码的具体作用。

```
# 使图表出现在文本中
%matplotlib inline
# 使用自己的图表样式文件
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

有关绘图的简单介绍

以上代码使用了一些技巧，使得图表更加美观。

首先，`%matplotlib inline` 是一个神奇的 Jupyter 笔记本命令，它使所有图表出现在笔记本中，而不是弹出新窗口。如果正在以交互方式运行 Jupyter 笔记本，那么你可以用 `%matplotlib notebook` 得到一张交互式图表，而不是每次绘图都得到一张静态图片。

其次，导入 matplotlib.pyplot，并指示它使用我们自己的绘图样式 `plt.style.use('style/elegant.mplstyle')`。在每章的第一次绘图前，你都会看到类似的代码。

或许你见过有人导入已有的样式，如 `plt.style.use('ggplot')`。但因为想要一些特别的设置，并使本书中的所有绘图都遵循同样的风格，所以我们使用自己的 Matplotlib 样式。如果想知道我们是如何做到的，可以查看本书库中的样式表文件：style/elegant.mplstyle。如果想了解关于样式的更多信息，可参见 Matplotlib 样式表文档。

现在我们回过头来绘制计数分布！

```
total_counts = np.sum(counts, axis=0) # 对列进行加总
                                      # （axis=1可以对行进行加总）
from scipy import stats

# 用高斯平滑估计密度
density = stats.kde.gaussian_kde(total_counts)

# 生成用来估计密度的值，准备绘图
x = np.arange(min(total_counts), max(total_counts), 10000)

# 生成密度图
fig, ax = plt.subplots()
ax.plot(x, density(x))
ax.set_xlabel("Total counts per individual")
ax.set_ylabel("Density")
plt.show()

print(f'Count statistics:\n min:  {np.min(total_counts)}'
      f'\n mean: {np.mean(total_counts)}'
      f'\n max: {np.max(total_counts)}')

Count statistics:
  min: 6231205
  mean: 52995255.33866667
  max: 103219262
```

可以看出，对于独立样本来说，最低的计数值与最高的计数值相差好几个数量级（见图 1-6）。这意味着每个独立样本生成的 RNA 测序 read 数都是不同的，我们称这些独立样本具有不同的库容量。

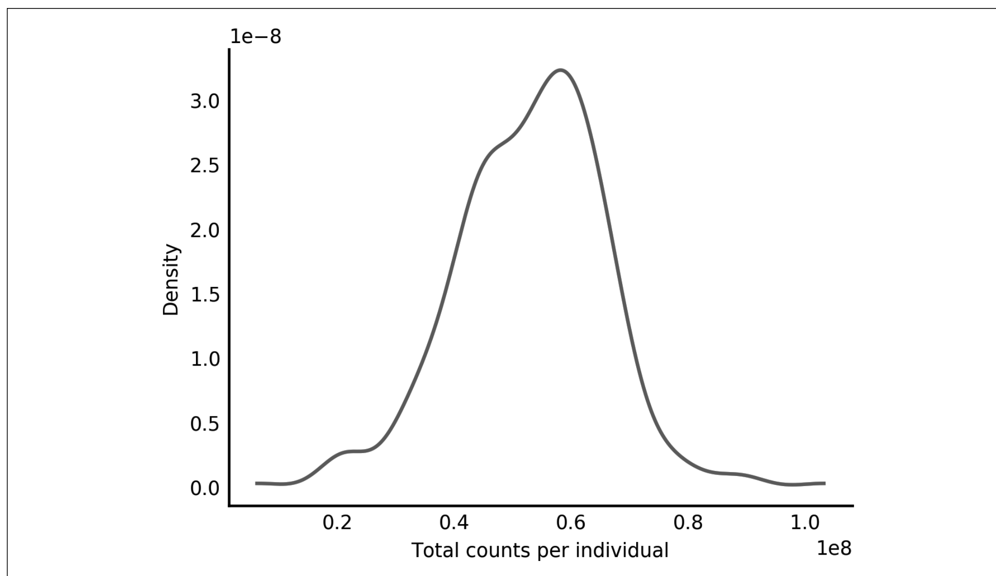


图 1-6：用 KDE 平滑生成的每个独立样本基因表达计数密度图

样本间的库容量标准化

我们来更加仔细地审视一下每个独立样本的基因表达范围，这样一来，对其进行标准化后，就可以更好地查看标准化的效果。我们将取出一个只有 70 列的随机样本子集，确保绘图不会过于凌乱。

```
# 取数据子集用于绘图
np.random.seed(seed=7) # 设定随机数种子，以得到一致的结果
# 随机选择70个样本
samples_index = np.random.choice(range(counts.shape[1]), size=70, replace=False)
counts_subset = counts[:, samples_index]

# 定制x轴标签，使图表更加易读
def reduce_xaxis_labels(ax, factor):
    """Show only every ith label to prevent crowding on x-axis
        e.g. factor = 2 would plot every second x-axis label,
        starting at the first.

    Parameters
    ----------
    ax : matplotlib plot axis to be adjusted
    factor : int, factor to reduce the number of x-axis labels by
    """
    plt.setp(ax.xaxis.get_ticklabels(), visible=False)
    for label in ax.xaxis.get_ticklabels()[factor-1::factor]:
        label.set_visible(True)

# 每个独立样本表达计数的箱线图
fig, ax = plt.subplots(figsize=(4.8, 2.4))

with plt.style.context('style/thinner.mplstyle'):
    ax.boxplot(counts_subset)
    ax.set_xlabel("Individuals")
    ax.set_ylabel("Gene expression counts")
    reduce_xaxis_labels(ax, 5)
```

基因表达的高端坐标附近明显有很多离群点，各个独立样本之间的波动也非常大，但图中很难看出这些情况，因为几乎所有点都聚集在 0 附近（见图 1-7）。因此，我们对数据进行 $\log(n + 1)$ 的对数变换，使其更易于查看（见图 1-8）。对数函数和 $n + 1$ 操作都可以用广播机制完成，这样一来，代码会更简洁，速度也会更快。

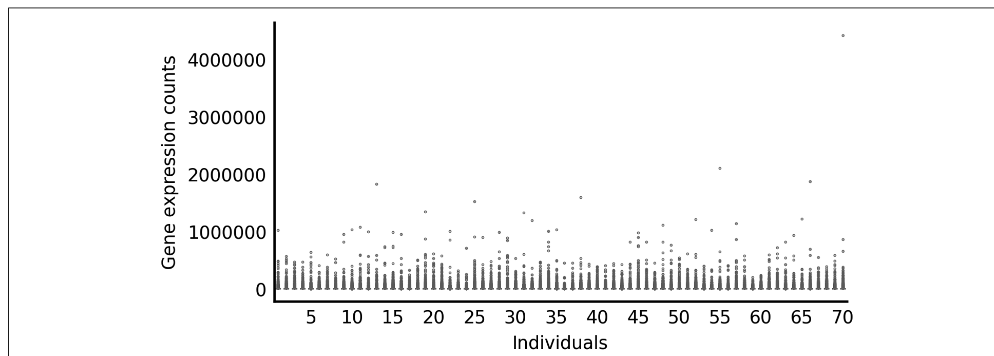


图 1-7：独立样本基因表达计数的箱线图

```
# 每个独立样本表达计数的箱线图
fig, ax = plt.subplots(figsize=(4.8, 2.4))

with plt.style.context('style/thinner.mplstyle'):
    ax.boxplot(np.log(counts_subset + 1))
    ax.set_xlabel("Individuals")
    ax.set_ylabel("log gene expression counts")
    reduce_xaxis_labels(ax, 5)
```

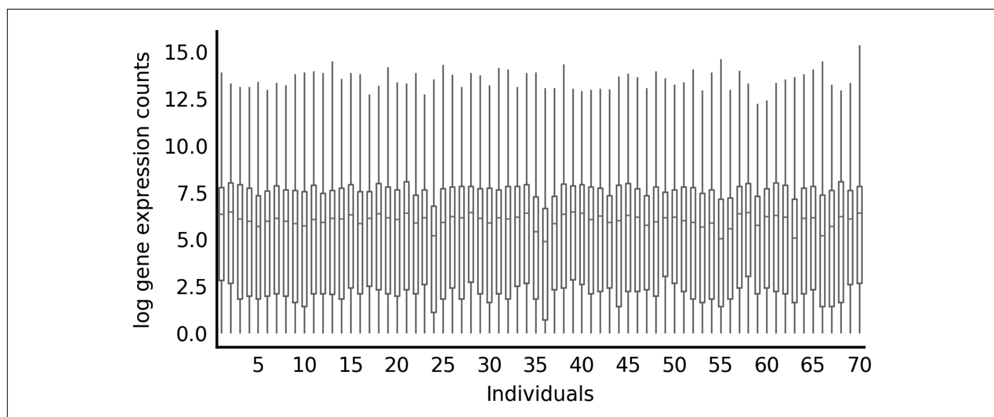


图 1-8：每个独立样本基因表达计数的箱线图（对数标度）

接下来看一下按库容量进行标准化会是什么情况（见图 1-9）。

```
# 按库容量进行标准化
# 用基因表达计数除以来自独立样本的总计数
# 再乘以1 000 000，使数值回到相似的量级
counts_lib_norm = counts / total_counts * 1000000
# 注意这里使用了两次广播机制
counts_subset_lib_norm = counts_lib_norm[:,samples_index]

# 每个独立样本表达计数的箱线图
fig, ax = plt.subplots(figsize=(4.8, 2.4))

with plt.style.context('style/thinner.mplstyle'):
    ax.boxplot(np.log(counts_subset_lib_norm + 1))
    ax.set_xlabel("Individuals")
    ax.set_ylabel("log gene expression counts")
    reduce_xaxis_labels(ax, 5)
```

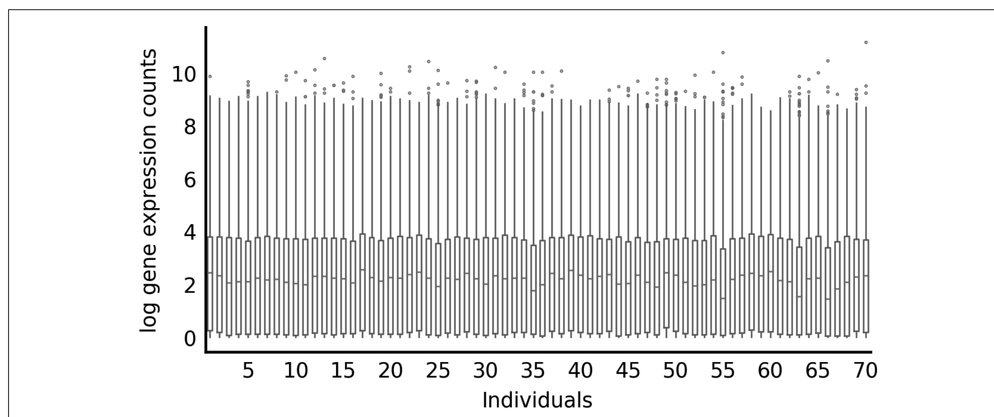


图 1-9：库容量标准化后的每个独立样本基因表达计数的箱线图（对数标度）

现在好多了！还需要注意的是，我们使用了两次广播机制。第一次是用所有基因表达计数除以那一列的总计数，第二次是将所有值都乘以 1 000 000。

最后，比较一下标准化后的数据和原始数据。

```
import itertools as it
from collections import defaultdict


def class_boxplot(data, classes, colors=None, **kwargs):
    """Make a boxplot with boxes colored according to the class they belong to.

    Parameters
    ----------
    data : list of array-like of float
        The input data. One boxplot will be generated for each element
        in `data`.
    classes : list of string, same length as `data`
        The class each distribution in `data` belongs to.

    Other parameters
    ----------------
    kwargs : dict
        Keyword arguments to pass on to `plt.boxplot`.
    """
    all_classes = sorted(set(classes))
    colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
    class2color = dict(zip(all_classes, it.cycle(colors)))

    # 将类映射到数据向量
    # 为了对齐，在类中没有数据的相应位置添加空列表
    class2data = defaultdict(list)
    for distrib, cls in zip(data, classes):
        for c in all_classes:
            class2data[c].append([])
```

```
        class2data[cls][-1] = distrib

    # 然后用适当的颜色依次生成每个箱线图
    fig, ax = plt.subplots()
    lines = []
    for cls in all_classes:
        # 为箱线图的所有元素设定颜色
        for key in ['boxprops', 'whiskerprops', 'flierprops']:
            kwargs.setdefault(key, {}).update(color=class2color[cls])
        # 画出箱线图
        box = ax.boxplot(class2data[cls], **kwargs)
        lines.append(box['whiskers'][0])
    ax.legend(lines, all_classes)
    return ax
```

现在我们可以根据标准化样本与未标准化样本绘制出带颜色的箱线图。出于演示的目的，每个类只显示出 3 个样本。

```
log_counts_3 = list(np.log(counts.T[:3] + 1))
log_ncounts_3 = list(np.log(counts_lib_norm.T[:3] + 1))
ax = class_boxplot(log_counts_3 + log_ncounts_3,
                   ['raw counts'] * 3 + ['normalized by library size'] * 3,
                   labels=[1, 2, 3, 1, 2, 3])
ax.set_xlabel('sample number')
ax.set_ylabel('log gene expression counts');
```

可以看出，考虑到库容量（样本分布的总和）后，样本的标准化分布更相似一些（见图 1-10）。现在我们是在样本之间进行同类比较，那么基因之间有什么样的差异呢？

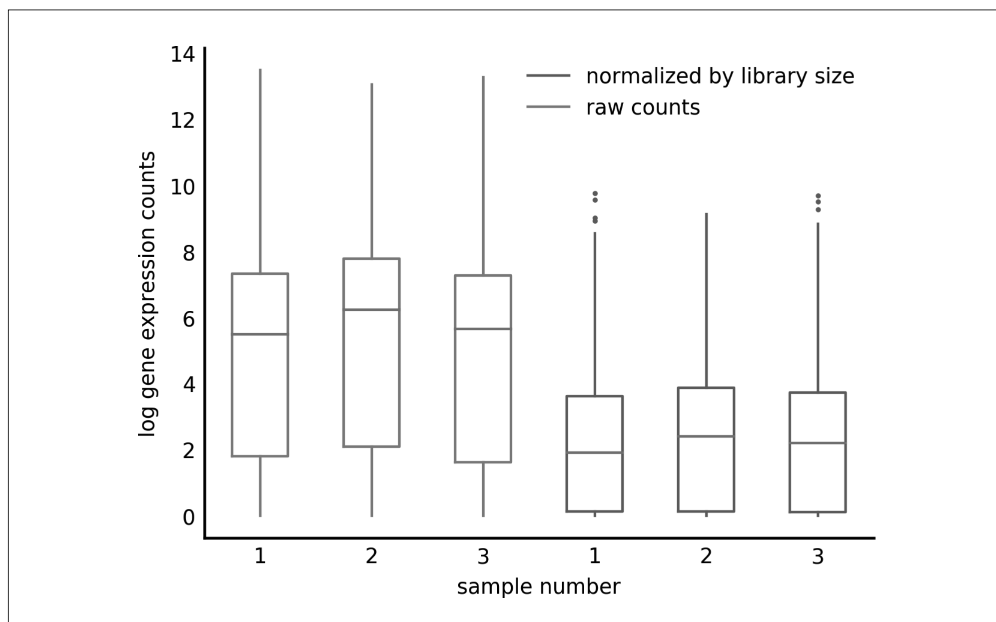


图 1-10：用 3 个样本比较原始计数与库容量标准化后的基因表达计数（对数标度）

1.4.2 基因间的标准化

当试图比较不同的基因时，我们同样会遇到麻烦。基因的计数值与基因长度相关。假设基因 *B* 的长度是基因 *A* 的两倍。二者在样本中的表达处于同一水平（也就是说，两个基因生成的 mRNA 分子数量非常接近）。回忆一下，在 RNA 测序实验中，我们将转录本分割成很多小的片段，然后从片段池中对 read 进行抽样。因此，如果一个基因的长度是另一个基因的两倍，那么它产生的片段数量也是另一个基因的两倍，被抽取出样本的概率也是另一个基因的两倍。因此我们可以预计基因 *B* 的表达计数是基因 *A* 的两倍（见图 1-11）。如果想要比较不同基因的表达水平，就必须做更多的标准化工作。

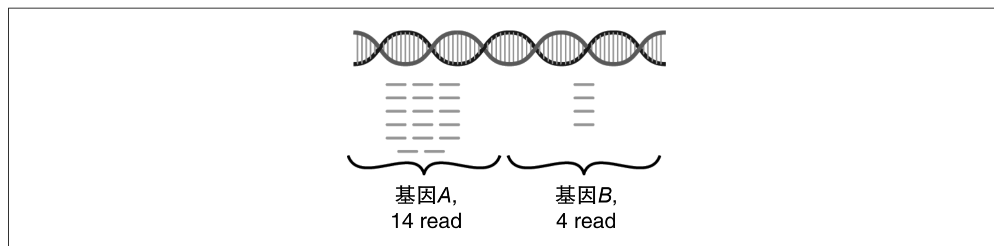


图 1-11：表达计数与基因长度之间的关系

我们看一下基因长度和表达计数间的关系能否体现在数据集中。首先，定义一个用于绘图的工具函数。

```
def binned_boxplot(x, y, *, # 看一下Python 3中特有的这个功能（*参见“Python 3小技巧”）
                   xlabel='gene length (log scale)',
                   ylabel='average log counts'):
    """Plot the distribution of `y` dependent on `x` using many boxplots.

    Note: all inputs are expected to be log-scaled.

    Parameters
    ----------
    x: 1D array of float
        Independent variable values.
    y: 1D array of float
        Dependent variable values.
    """
    # 根据观测密度定义x的分箱
    x_hist, x_bins = np.histogram(x, bins='auto')

    # 用np.digitize为分箱标号
    # 丢弃最后一个分箱的边缘，因为它违背了digitize的右开放假设。
    # 最大的观测正确地进入最后一个分箱
    x_bin_idxs = np.digitize(x, x_bins[:-1])

    # 用这些标号创建一个数组列表，其中每个数组都包含那个分箱中的x所对应的y值。
    # 这是plt.boxplot所期望的输入格式。
    binned_y = [y[x_bin_idxs == i]
                for i in range(np.max(x_bin_idxs))]
    fig, ax = plt.subplots(figsize=(4.8,1))
```

```
# 用分箱中心作为x轴的标签
x_bin_centers = (x_bins[1:] + x_bins[:-1]) / 2
x_ticklabels = np.round(np.exp(x_bin_centers)).astype(int)

# 生成箱线图
ax.boxplot(binned_y, labels=x_ticklabels)

# 仅显示每10个标签，以免x轴过于拥挤
reduce_xaxis_labels(ax, 10)

# 调整坐标轴名称
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel);
```

Python 3 小技巧：用 * 创建强制关键字参数

从 3.0 版开始，Python 允许使用“强制关键字”参数。这些参数必须用关键字来调用，不能只靠位置。例如，要想调用 binned_boxplot，可以使用以下形式。

```
>>> binned_boxplot(x, y, xlabel='my x label', ylabel='my y label')
```

但不能使用以下形式，虽然它在 Python 2 中有效，但在 Python 3 中会引发错误。

```
>>> binned_boxplot(x, y, 'my x label', 'my y label')

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-7a118d2d5750in <module>()
    1 x_vals = [1, 2, 3, 4, 5]
    2 y_vals = [1, 2, 3, 4, 5]
----3 binned_boxplot(x, y, 'my x label', 'my y label')

TypeError: binned_boxplot() takes 2 positional arguments but 4 were given
```

这种设计理念是为了防止你意外地写成以下形式。

```
binned_boxplot(x, y, 'my y label')
```

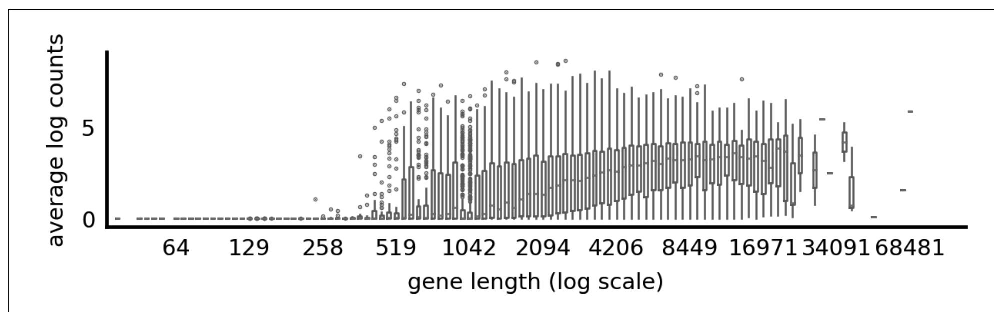
这会导致在 x 轴上设定 y 轴标签，当使用很多没有明显顺序的可选参数时，经常会出现这种错误。

现在计算一下基因长度和表达计数。

```
log_counts = np.log(counts_lib_norm + 1)
mean_log_counts = np.mean(log_counts, axis=1) # 对所有样本
log_gene_lengths = np.log(gene_lengths)

with plt.style.context('style/thinner.mplstyle'):
    binned_boxplot(x=log_gene_lengths, y=mean_log_counts)
```

从下图中可以看出，基因越长，测量出的计数越多。正如前面解释过的，这是技术手段造成的，并不是一种生物学现象！怎么解决这个问题呢？



1.4.3 样本与基因标准化：RPKM

对 RNA 测序数据进行标准化的最简单的方法之一就是 RPKM：每百万 read 中来自每千碱基转录本的 read 数。RPKM 综合了按照样本和按照基因进行标准化的思想。当计算 RPKM 时，我们既对库容量（每列之和）也对基因长度进行标准化。

为了理解 RPKM 是如何推导出的，我们先定义以下几个值：

- C= 映射到一个基因的 read 数
- L= 一个基因以碱基对为单位的外显子长度
- N= 实验中映射 read 的总数

首先，计算每千碱基的 read 数。

每碱基 read 数如下所示：

$$\frac{C}{L}$$

公式需要的是每千碱基 read 数，而不是每碱基 read 数。1 千碱基 = 1000 碱基，因此我们需要将长度（L）除以 1000。

每千碱基 read 数如下所示：

$$\frac{C}{L/1000}=\frac{10^3C}{L}$$

接着需要按库容量进行标准化。如果只是除以映射 read 数，可以得到：

$$\frac{10^3C}{LN}$$

但是生物学家喜欢用每百万 read 数，这样最后的值不会太大。按照每百万 read 数计算，可以得到：

$$\frac{10^3C}{L(N/10^6)}=\frac{10^9C}{LN}$$

归纳一下，要想计算每百万 read 中来自每千碱基转录本的 read 数，可以使用以下公式：

$$\text{RPKM}=\frac{10^9 C}{LN}$$

现在在整个计数数组上实现 RPKM。

```
# 使变量名称与RPKM公式一致，以便比较
C = counts
N = counts.sum(axis=0) # 对每列进行加总，以得到每个样本的总read数
L = gene_lengths # 每个基因的长度，与C中的行匹配
```

首先要乘以 10^9。因为计数（C）是个 N 维数组，所以可以使用广播机制。如果用单个值乘以 N 维数组，那么这个值就可以广播到整个数组中。

```
# 所有计数乘以10⁹
C_tmp = 10^9 * C
```

然后要除以基因长度。将单一值广播到二维数组非常简单，只要用这个值乘以数组中的每个元素就可以了。但如果需要将二维数组除以一个一维数组，应该怎么做呢？

1. 广播规则

广播机制允许在不同形状的 N 维数组之间进行计算。NumPy 通过广播规则使得这种操作更容易一些。当两个数组具有同样数量的维度时，如果每个维度大小匹配或者有一个维度等于 1，那么就可以进行广播。如果两个数组具有不同数量的维度，那么就可以在维度较少的数组前面添加 `(1,)`，直到维度数量相等，然后再使用标准广播规则。

举例来说，假设有两个 N 维数组 A 和 B，形状分别是 `(5,2)` 和 `(2,)`，我们用广播规则定义 A 和 B 的积 `A * B`。因为 B 的维度数比 A 少，所以在计算过程中在 B 前面添加一个值为 1 的新维度，使 B 的形状变为 `(1,2)`。最后，只要 B 的形状与 A 不匹配，就不断对 B 进行**堆积复制**，直到它的形状变为 `(5,2)`。这个过程是“虚拟”进行的，并不占用任何额外的内存。在做乘法时，两个数组的各个元素依次相乘，最后的结果是一个与 A 形状相同的数组。

假设我们有另一个形状为 `(2,5)` 的数组 C。要想使 C 与 B 相乘（或相加），可以试着在 B 的形状前面添加 `(1,)`，但这么做的话，最后的形状还是不兼容：`(2,5)` 和 `(1,2)`。如果想对数组进行广播，就必须手动在 B 的后面添加一个维度，使它们的形状变为 `(2,5)` 和 `(2,1)`，这样就可以进行广播了。

在 NumPy 中，可以用 `np.newaxis` 显式地向 B 中添加一个新维度。接下来看看如何在 RPKM 标准化中进行这种操作。

先看一下数组的维度。

```
print('C_tmp.shape', C_tmp.shape)
print('L.shape', L.shape)

C_tmp.shape (20500, 375)
L.shape (20500,)
```

可以看到，`C_tmp` 有两个维度，而 `L` 只有一个。因此，一个额外的维度会在广播过程中添

加到 L 的前面，然后可以得到：

```
C_tmp.shape (20500, 375)
L.shape (1, 20500)
```

维度还是不匹配！我们要在 C_tmp 的第一个维度上对 L 进行广播，因此需要自己来调整 L 的维度。

```
L = L[:, np.newaxis] # 在L的后面添加一个值为1的维度
print('C_tmp.shape', C_tmp.shape)
print('L.shape', L.shape)

C_tmp.shape (20500, 375)
L.shape (20500, 1)
```

这样维度就匹配了或者其中一个等于 1，我们可以进行广播了。

```
# 将每一行除以该行基因的长度（L）
C_tmp = C_tmp / L
```

最后，还需要按照库容量进行标准化。库容量就是一列的计数值总和。回忆一下，我们已经用以下代码计算出了 N：

```
N = counts.sum(axis=0) # sum each column to get total reads per sample

# 检查C_tmp和N的形状
print('C_tmp.shape', C_tmp.shape)
print('N.shape', N.shape)

C_tmp.shape (20500, 375)
N.shape (375,)
```

一旦触发了广播机制，一个额外的维度就会被添加到 N 的前面。

```
N.shape (1, 375)
```

维度是匹配的，因此我们无须做任何工作。但为了更具可读性，可以显式地为 N 添加一个新维度。

```
# 将每一列除以该列的总计数（N）
N = N[np.newaxis, :]
print('C_tmp.shape', C_tmp.shape)
print('N.shape', N.shape)

C_tmp.shape (20500, 375)
N.shape (1, 375)

# 将每一列除以该列的总计数（N）
rpkm_counts = C_tmp / N
```

我们将这个过程写成一个函数，以便重用。

```
def rpkm(counts, lengths):
    """Calculate reads per kilobase transcript per million reads.

    RPKM = (10^9 * C) / (N * L)
```

```
    Where:
    C = Number of reads mapped to a gene
    N = Total mapped reads in the experiment
    L = Exon length in base pairs for a gene

    Parameters
    ----------
    counts: array, shape (N_genes, N_samples)
        RNAseq (or similar) count data where columns are individual samples
        and rows are genes.
    lengths: array, shape (N_genes,)
        Gene lengths in base pairs in the same order
        as the rows in counts.

    Returns
    -------
    normed : array, shape (N_genes, N_samples)
        The RPKM normalized counts matrix.
    """
    N = np.sum(counts, axis=0) # 对每列进行加总，得到每个样本的总read数
    L = lengths
    C = counts

    normed = 1e9 * C / (N[np.newaxis, :] * L[:, np.newaxis])

    return(normed)

counts_rpkm = rpkm(counts, gene_lengths)
```

2. 基因间的RPKM标准化

我们看一下 RPKM 标准化的实际效果。首先需要说明一下，以下是表达计数的对数均值分布，它是基因长度的一个函数（见图 1-12）。

```
log_counts = np.log(counts + 1)
mean_log_counts = np.mean(log_counts, axis=1)
log_gene_lengths = np.log(gene_lengths)

with plt.style.context('style/thinner.mplstyle'):
    binned_boxplot(x=log_gene_lengths, y=mean_log_counts)
```

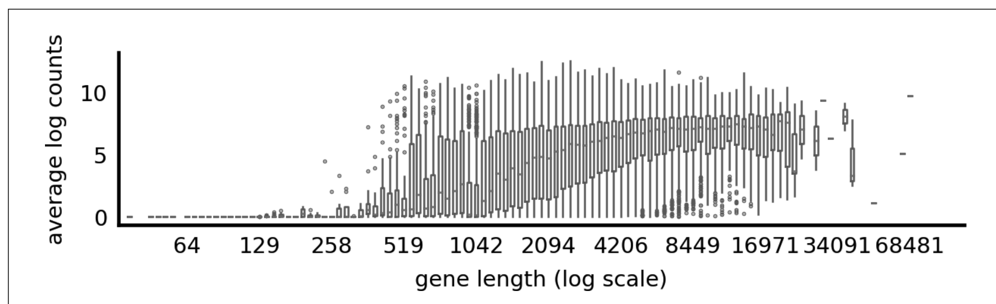
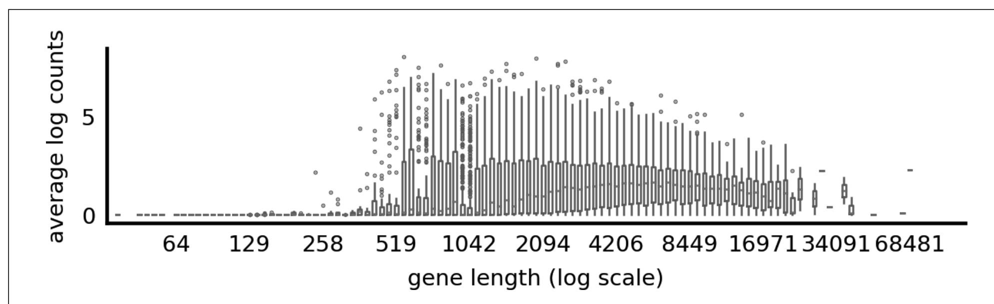


图 1-12：RPKM 标准化前的基因长度与表达计数均值之间的关系（对数标度）

现在用 RPKM 标准化后的值绘制同样的图形。

```
log_counts = np.log(counts_rpkm + 1)
mean_log_counts = np.mean(log_counts, axis=1)
log_gene_lengths = np.log(gene_lengths)

with plt.style.context('style/thinner.mplstyle'):
    binned_boxplot(x=log_gene_lengths, y=mean_log_counts)
```



从图中可以看出，表达计数均值平整了一些，尤其对于那些长度大于 3000 个碱基对的基因来说。（较短的基因仍然表现出比较低的表达水平——可能它们太短了，RPKM 方法在统计上对其没有什么效果。）

RPKM 标准化对于比较不同基因的表达水平非常有用。我们已经看到了，基因越长，其表达计数就越高，但这并不意味着表达水平实际更高。为了说明这句话的含义，我们选择一个短基因和一个长基因，比较一下它们在 RPKM 标准化前后的计数。

```
gene_idxs = np.array([80, 186])
gene1, gene2 = gene_names[gene_idxs]
len1, len2 = gene_lengths[gene_idxs]
gene_labels = [f'{gene1}, {len1}bp', f'{gene2}, {len2}bp']

log_counts = list(np.log(counts[gene_idxs] + 1))
log_ncounts = list(np.log(counts_rpkm[gene_idxs] + 1))

ax = class_boxplot(log_counts,
                   ['raw counts'] * 3,
                   labels=gene_labels)
ax.set_xlabel('Genes')
ax.set_ylabel('log gene expression counts over all samples');
```

如果只看原始计数，似乎长基因 TXNDC5 比短基因 RPL24 的表达水平稍高一些（见图 1-13）。但进行 RPKM 标准化后，图形就发生了改变。

```
ax = class_boxplot(log_ncounts,
                   ['RPKM normalized'] * 3,
                   labels=gene_labels)
ax.set_xlabel('Genes')
ax.set_ylabel('log RPKM gene expression counts over all samples');
```

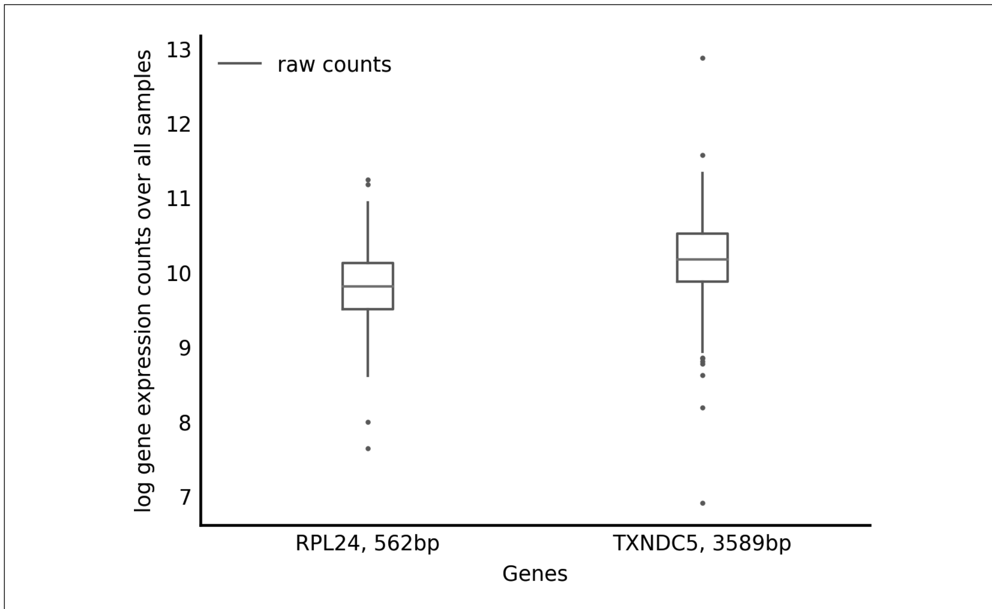


图 1-13：比较 RPKM 标准化前两个基因的表达水平

现在 RPL24 的表达水平看起来比 TXNDC5 明显高出很多（见图 1-14）。这是因为 RPKM 方法中包含了对基因长度的标准化，所以我们可以在不同长度的基因之间直接进行比较。

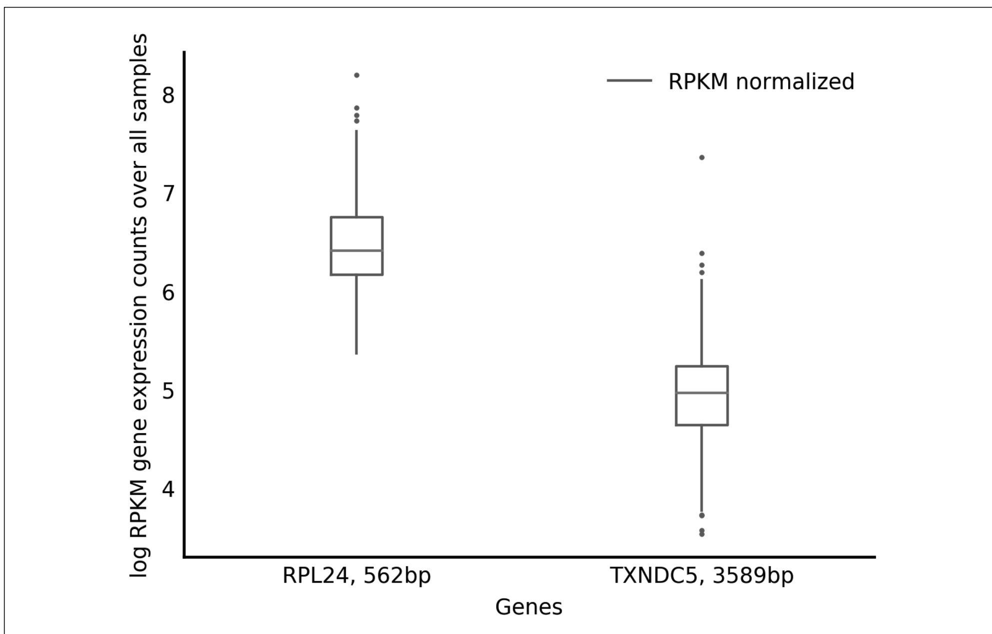


图 1-14：比较 RPKM 标准化后两个基因的表达水平

1.5 小结

至此，我们学习了以下内容：

- 用 pandas 导入数据
- 熟悉 NumPy 的核心对象类——*N* 维数组
- 用强大的广播机制使得计算过程更加优雅

第 2 章将继续用同样的数据集来实现一种更加复杂的标准化技术，然后用聚类技术预测皮肤癌患者的死亡率。

第2章

用NumPy和SciPy进行分位数标准化

如果一开始不能理解空间国的深刻奥秘，也不要垂头丧气，慢慢你就会明白的。

——Edwin A. Abbott，《平面国：一个多维的传奇故事》

本章将继续分析第 1 章中的基因表达数据，但目的略有不同：我们想要用每名患者的**基因表达档案**（基因表达测量的完整向量）来预测他们的预期存活率。为了使用完整的档案，我们需要一种比第 1 章中的 RPKM 更强大的标准化方法。我们将执行**分位数标准化**，这是一种可以确保测量结果符合特定分布的技术。这种方法遵从一个严格的假设：如果数据没有按照需要的形状分布，就强制它符合所需形状！这听起来有点像作弊，但人们已经证明，在具体分布影响不大，但总体中值的相对改变非常重要的情况下，这是一种简单而有效的方法。例如，根据 Bolstad 及其同事的研究，这种方法在从 DNA 微阵列数据中发现已知表达水平的工作中表现得非常突出。

在学习本章的过程中，我们将为 TCGA 计划的论文“Genomic Classification of Cutaneous Melanoma”中的图 5A 和图 5B 制作一份简化版本。

在实现分位数标准化时，我们要有效地用 NumPy 和 SciPy 编写一个快速、高效且优雅的函数。分位数标准化包括以下 3 个步骤：

(1) 对每列的值进行排序；
(2) 找出每个结果行的平均值；
(3) 用平均值列的分位数替换每列的分位数。

```
import numpy as np
from scipy import stats

def quantile_norm(X):
    """Normalize the columns of X to each have the same distribution.
```

```
    Given an expression matrix (microarray data, read counts, etc) of M genes
    by N samples, quantile normalization ensures all samples have the same
    spread of data (by construction).

    The data across each row are averaged to obtain an average column. Each
    column quantile is replaced with the corresponding quantile of the average
    column.

    Parameters
    ----------
    X : 2D array of float, shape (M, N)
        The input data, with M rows (genes/features) and N columns (samples).

    Returns
    -------
    Xn : 2D array of float, shape (M, N)
        The normalized data.
    """
    # 计算分位数
    quantiles = np.mean(np.sort(X, axis=0), axis=1)

    # 计算每列的秩次。将每个观测替换为其在列中的秩次：最小的观测替换为1，
    # 第二小的观测替换为2，以此类推，最大的观测替换为M，即行的数量
    ranks = np.apply_along_axis(stats.rankdata, 0, X)

    # 将秩次转换为0~M-1的整数标号
    rank_indices = ranks.astype(int) - 1

    # 以秩次矩阵中的每个秩次为索引，返回分位数相应位置的值
    Xn = quantiles[rank_indices]

    return(Xn)
```

在实际操作中，由于基因表达计数数据之间的巨大差异，经常需要在分位数标准化前对数据进行对数变换。为此，我们编写了一个辅助函数来进行对数变换。

```
def quantile_norm_log(X):
    logX = np.log(X + 1)
    logXn = quantile_norm(logX)
    return logXn
```

这两个函数综合说明了使得 NumPy 威力巨大的多种因素（你应该记得第 1 章中体现了前三种因素）。

- 数组可以是一维的（如列表）、二维的（如矩阵），也可以是更高维度的。这使得它们可以表示多种类型的数值型数据。我们的示例表示的就是一个二维矩阵。
- 数组可以同时进行多个数值操作。在 `quantile_norm_log` 的第一行中，我们在一次调用中就为 `X` 的每个值都加了 1 并取了对数，这称为**向量化**。
- 数组可以沿着**轴**进行操作。在 `quantile_norm` 的第一行中，我们为 `np.sort` 指定了 `axis` 参数，沿着每一列对数据进行排序。然后又通过指定一个**不同的** `axis` 沿着每一行计算数据的均值。

- 数组奠定了 Python 科学生态系统的基础。`scipy.stats.rankdata` 函数不是在 Python 列表上进行操作，而是在 NumPy 数组上。很多 Python 科学库都是这样的。
- 即使函数不具备 `axis=` 关键字参数，也可以通过 NumPy 的 `apply_along_axis` 函数沿着轴进行操作。
- 数组可以通过**花式索引**（fancy indexing）支持多种数据操作：`Xn = quantiles[ranks]`。这可能是 NumPy 中最难以理解的部分，但同时也是最有用的部分之一。接下来将对其做进一步介绍。

2.1 获取数据

与第 1 章一样，我们将处理 TCGA 皮肤癌 RNA 测序数据集。我们的目标是用皮肤癌患者的 RNA 表达数据来预测他们的死亡率。正如前面提过的，到本章末尾我们将重新生成 TCGA 组织的一篇论文中的图 5A 和图 5B 的简化版本。

与第 1 章一样，首先要用 pandas 更加轻松地读取数据。先将计数数据读取到一个 pandas 表格中。

```
import numpy as np
import pandas as pd

# 导入TCGA黑色素瘤数据
filename = 'data/counts.txt'
data_table = pd.read_csv(filename, index_col=0) # 用pandas解析文件

print(data_table.iloc[:5, :5])
```

```
       00624286-41dd-476f-a63b-d2a5f484bb45  TCGA-FS-A1Z0  TCGA-D9-A3Z1 \
A1BG                                1272.36        452.96        288.06
A1CF                                   0.00          0.00          0.00
A2BP1                                  0.00          0.00          0.00
A2LD1                                164.38        552.43        201.83
A2ML1                                 27.00          0.00          0.00

       02c76d24-f1d2-4029-95b4-8be3bda8fdbe  TCGA-EB-A51B
A1BG                                 400.11        420.46
A1CF                                   1.00          0.00
A2BP1                                  0.00          1.00
A2LD1                                165.12         95.75
A2ML1                                  0.00          8.00
```

通过查看 `data_table` 的行与列，可以知道列是样本，行是基因。现在将计数数据放在 NumPy 数组中。

```
# 包含每个独立样本中的每个基因表达计数的二维数组
counts = data_table.values
```

2.2 独立样本间的基因表达分布差异

现在，我们通过绘制每个独立样本的计数分布来感受一下计数数据。我们将用高斯核函数

对数据的波动进行平滑处理，以更好地认识数据的整体形状。

像往常一样，先设定绘图风格。

```
# 使图表出现在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

接着编写一个绘图函数，用 SciPy 的 gaussian_kde 函数绘制出平滑的数据分布。

```
from scipy import stats

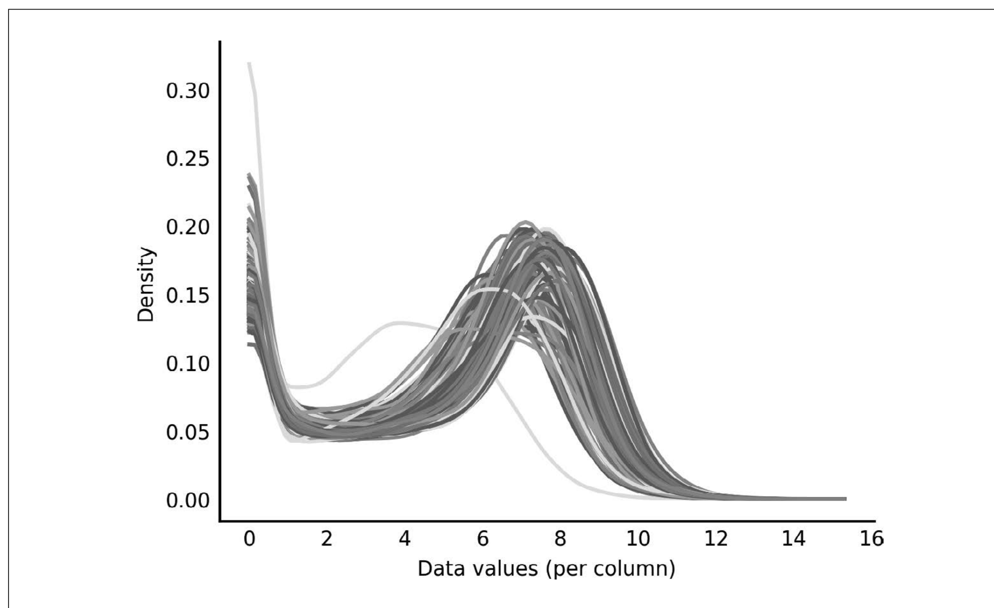
def plot_col_density(data):
    """For each column, produce a density plot over all rows."""

    # 用高斯平滑来估计密度
    density_per_col = [stats.gaussian_kde(col) for col in data.T]
    x = np.linspace(np.min(data), np.max(data), 100)

    fig, ax = plt.subplots()
    for density in density_per_col:
        ax.plot(x, density(x))
    ax.set_xlabel('Data values (per column)')
    ax.set_ylabel('Density')
```

现在可以用这个函数绘制原始数据的分布了，先不进行任何标准化。绘图如下所示。

```
# 标准化前
log_counts = np.log(counts + 1)
plot_col_density(log_counts)
```



可以看出，尽管这些计数分布大体上相似，但有些独立样本的分布形状非常漂亮，有些则歪七扭八。实际上，考虑到这张图使用的是对数标度，这些分布的峰值位置实际相差好几个数量级！在本章后面分析计数数据时，我们会假设基因表达中的变化是由样本间的生物学差别引起的。但像图中这样明显的分布变动则表明差异是由技术手段造成的，也就是说，基因表达中的变化很可能是由样本处理方式的差异造成的，而不是因为样本在生物学上有差别。因此，我们将对其进行标准化，以消除样本间的这些整体差异。

正如本章开头所说，为了进行这种标准化，需要执行分位数标准化。分位数标准化的思想是，所有样本都应该具有相似的分布，因此形状上的任何差别都应归结于技术差异。更正式的表述是，给定一个形状为 (n_genes, n_samples) 的表达矩阵（微阵列数据、read 计数等），分位数标准化可以通过数据构建确保所有样本（列）具有相同的数据分布。

可以用 NumPy 和 SciPy 非常轻松、高效地实现分位数标准化。简要概括一下，以下就是本章开头介绍过的分位数标准化的实现代码。

假设 ***X*** 是我们的输入矩阵。

```
import numpy as np
from scipy import stats

def quantile_norm(X):
    """Normalize the columns of X to each have the same distribution.

    Given an expression matrix (microarray data, read counts, etc.) of M genes
    by N samples, quantile normalization ensures all samples have the same
    spread of data (by construction).

    The data across each row are averaged to obtain an average column. Each
    column quantile is replaced with the corresponding quantile of the average
    column.

    Parameters
    ----------
    X : 2D array of float, shape (M, N)
        The input data, with M rows (genes/features) and N columns (samples).

    Returns
    -------
    Xn : 2D array of float, shape (M, N)
        The normalized data.
    """
    # 计算分位数
    quantiles = np.mean(np.sort(X, axis=0), axis=1)

    # 计算每列的秩次，将每个观测替换为其在列中的秩次：
    # 最小的观测替换为1，第二小的观测替换为2，以此类推，
    # 最大的观测替换为M，即行的数量
    ranks = np.apply_along_axis(stats.rankdata, 0, X)

    # 将秩次转换为0~M-1的整数标号
    rank_indices = ranks.astype(int) - 1
```

```
    # 以秩次矩阵中的每个秩次为索引，返回分位数相应位置的值
    Xn = quantiles[rank_indices]

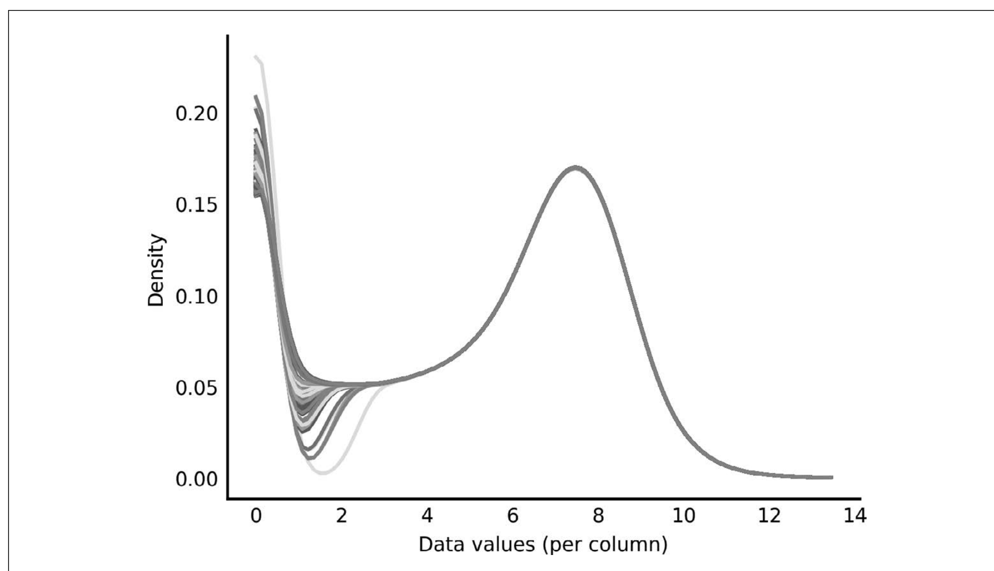
    return(Xn)

def quantile_norm_log(X):
    logX = np.log(X + 1)
    logXn = quantile_norm(logX)
    return logXn
```

现在我们看一下分位数标准化后的分布是什么形状。效果如下图所示。

```
# 标准化后
log_counts_normalized = quantile_norm_log(counts)

plot_col_density(log_counts_normalized)
```



不出所料，这些分布现在看上去几乎完全一致！[分布左尾上的差别是由各列数据中低计数值（如 0、1、2，等等）的不同数量造成的。]

我们已经对计数数据进行了标准化，可以用基因表达数据来预测患者的情况了。

2.3 计数数据的双向聚类

对样本进行聚类可以告诉我们哪些样本具有相似的基因表达档案，这表明了这些样本在其他方面也可能具有相似的特性。对数据进行标准化后，就可以对表达矩阵的基因（行）和样本（列）进行聚类。对行进行聚类可以告诉我们哪些基因的表达值是有联系的，这表明它们在本次研究阶段中是共同工作的。**双向聚类**意味着同时在数据的行和列上进行聚类。通过对行进行聚类，可以找出那些共同工作的基因；通过对列进行聚类，可以找出哪些样本是相似的。

因为聚类是一项开销巨大的操作，所以我们将分析限制在 1500 个差异最大的基因上，这些基因可以代表任意维度上的绝大多数相关性信号。

```
def most_variable_rows(data, *, n=1500):
    """Subset data to the n most variable rows

    In this case, we want the n most variable genes.

    Parameters
    ----------
    data : 2D array of float
        The data to be subset
    n : int, optional
        Number of rows to return.

    Returns
    -------
    variable_data : 2D array of float
        The `n` rows of `data` that exhibit the most variance.
    """
    # 沿列轴计算方差
    rowvar = np.var(data, axis=1)
    # 得到排序后的行号（升序），取最后n个
    sort_indices = np.argsort(rowvar)[-n:]
    # 用作数据索引
    variable_data = data[sort_indices, :]
    return variable_data
```

接下来要用一个函数对数据进行双向聚类。通常来说，你需要用 scikit-learn 库中的一个复杂聚类算法来进行聚类。就我们的示例而言，为了简单和易于展示，我们想要使用层次聚类。SciPy 库中正好有一个非常完美的层次聚类模块，但你需要仔细思考一下才能理解其接口。

提示一下，层次聚类是一种用逐渐形成的簇对观测进行分组的方法。最初，每个观测本身都是一个簇。然后，距离最近的两个簇会合并，并不断进行这样的合并，直至所有观测都在一个簇中。这种连续合并形成了一棵**合并树**。通过将树修剪成特定高度，可以得到一个粒度更细或更粗的观测聚类。

`scipy.cluster.hierarchy` 中的 `linkage` 函数可以对矩阵的行执行层次聚类，它用特定的度量方式（如欧氏距离、曼哈顿距离等）和测距方法（如两个簇中所有观测间的平均距离）来测量两个簇之间的距离。

它以“联系矩阵”的方式返回合并树，这个矩阵包含了每次的合并操作以及合并时计算出的距离，还有结果簇中的观测数量。`linkage` 函数的文档中有以下内容：

> 一个标号小于 *n* 的簇对应于 *n* 个初始观测中的一个。簇 `Z[i, 0]` 和 `Z[i, 1]` 之间的距离由 `Z[i, 2]` 给出。第四个值 `Z[i, 3]` 表示新形成的簇中的初始观测的数量。

哇！信息量好大，让我们立即行动起来，希望你能尽快掌握代码中的诀窍。首先要定义一个函数 `bicluster`，该函数既对矩阵中的行也对矩阵中的列进行聚类。

```
from scipy.cluster.hierarchy import linkage


def bicluster(data, linkage_method='average', distance_metric='correlation'):
    """Cluster the rows and the columns of a matrix.

    Parameters
    ----------
    data : 2D ndarray
        The input data to bicluster.
    linkage_method : string, optional
        Method to be passed to `linkage`.
    distance_metric : string, optional
        Distance metric to use for clustering. See the documentation
        for ``scipy.spatial.distance.pdist`` for valid metrics.

    Returns
    -------
    y_rows : linkage matrix
        The clustering of the rows of the input data.
    y_cols : linkage matrix
        The clustering of the cols of the input data.
    """
    y_rows = linkage(data, method=linkage_method, metric=distance_metric)
    y_cols = linkage(data.T, method=linkage_method, metric=distance_metric)
    return y_rows, y_cols
```

小菜一碟：我们只是对输入矩阵调用了 `linkage` 函数，并对矩阵进行了**转置**，这样矩阵的列就变成了行，行变成了列。

2.4 簇的可视化

接着我们定义一个函数对聚类结果进行可视化。我们将重新排列输入数据的行和列，使相似的行排在一起，相似的列也排在一起。此外，我们还将表示出行和列的合并树，显示哪些观测属于哪个簇。合并树以树状图的样式呈现，分支长度表示观测间的相似程度（长度越短表示越相似）。

注意，以下程序中的很多参数都是用硬编码的方式写成的。绘图时很难避免这种方式，而且图的设计经常需要目测出合适的比例。

```
from scipy.cluster.hierarchy import dendrogram, leaves_list


def clear_spines(axes):
    for loc in ['left', 'right', 'top', 'bottom']:
        axes.spines[loc].set_visible(False)
    axes.set_xticks([])
    axes.set_yticks([])


def plot_bicluster(data, row_linkage, col_linkage,
                   row_nclusters=10, col_nclusters=3):
```

```
"""Perform a biclustering, plot a heatmap with dendrograms on each axis.

Parameters
----------
data : array of float, shape (M, N)
    The input data to bicluster.
row_linkage : array, shape (M-1, 4)
    The linkage matrix for the rows of `data`.
col_linkage : array, shape (N-1, 4)
    The linkage matrix for the columns of `data`.
row_nclusters, col_nclusters : int, optional
    Number of clusters for rows and columns.
"""
fig = plt.figure(figsize=(4.8, 4.8))

# 计算并绘制行方向的树状图
# add_axes接受一个“矩形”输入，向基础图中添加一个子图
# 可以认为基础图每边的边长都是1，左下角位于(0, 0)
# 传递给add_axes的参数是子图的相对尺寸，分别为左侧、底部、宽度和高度
# 因此，为了画出左侧的树状图（行方向），我们要创建一个矩形，与基础图相比，
# 其左下角位于(0.09, 0.1)，宽度是0.2，高度是0.6
ax1 = fig.add_axes([0.09, 0.1, 0.2, 0.6])
# 对于给定数目的簇，通过查看联系矩阵中的相应距离标注，可以得到一个联系树的剪枝版本
threshold_r = (row_linkage[-row_nclusters, 2] +
               row_linkage[-row_nclusters+1, 2]) / 2
with plt.rc_context({'lines.linewidth': 0.75}):
    dendrogram(row_linkage, orientation='left',
               color_threshold=threshold_r, ax=ax1)
clear_spines(ax1)

# 计算并绘制列方向的树状图
# 参见上面对add_axes参数的解释
ax2 = fig.add_axes([0.3, 0.71, 0.6, 0.2])
threshold_c = (col_linkage[-col_nclusters, 2] +
               col_linkage[-col_nclusters+1, 2]) / 2
with plt.rc_context({'lines.linewidth': 0.75}):
    dendrogram(col_linkage, color_threshold=threshold_c, ax=ax2)
clear_spines(ax2)

# 绘制数据热图
ax = fig.add_axes([0.3, 0.1, 0.6, 0.6])

# 按照树状图的叶子节点对数据排序
idx_rows = leaves_list(row_linkage)
data = data[idx_rows, :]
idx_cols = leaves_list(col_linkage)
data = data[:, idx_cols]

im = ax.imshow(data, aspect='auto', origin='lower', cmap='YlGnBu_r')
clear_spines(ax)

# 坐标轴标签
ax.set_xlabel('Samples')
ax.set_ylabel('Genes', labelpad=125)
```

```
    # 绘制图例
    axcolor = fig.add_axes([0.91, 0.1, 0.02, 0.6])
    plt.colorbar(im, cax=axcolor)

    # 显示图形
    plt.show()
```

现在我们将这些函数应用于标准化后的计数矩阵，以显示行和列的聚类（见图 2-1）。

```
counts_log = np.log(counts + 1)
counts_var = most_variable_rows(counts_log, n=1500)
yr, yc = bicluster(counts_var, linkage_method='ward',
                   distance_metric='euclidean')
with plt.style.context('style/thinner.mplstyle'):
    plot_bicluster(counts_var, yr, yc)
```

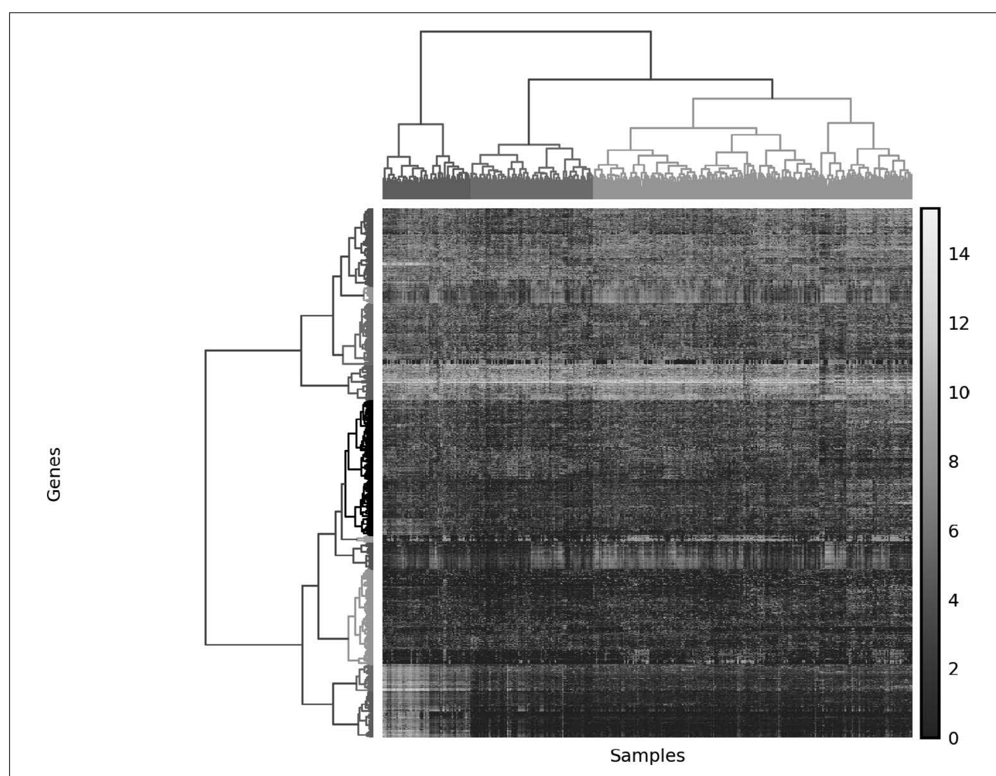


图 2-1：这张热图展示了所有样本和基因间的基因表达水平。颜色表示基因表达水平。行和列按照我们的聚类结果进行分组。我们可以沿着 *y* 轴查看基因簇，在 *x* 轴上方查看样本簇

2.5 预测幸存者

我们可以看到，样本数据自然地落在了至少两个簇中，也可能是三个。这些簇有实际意义吗？为了回答这个问题，我们需要使用患者数据；可以从上述论文的数据仓库中得到这些

数据。进行一些预处理后，可以得到一张患者表格，其中包含了每名患者的存活信息。然后我们可以将这些信息与计数的聚类结果进行匹配，弄清楚是否可以用患者的基因表达来预测他们的病理学差异。

```
patients = pd.read_csv('data/patients.csv', index_col=0)
patients.head()
```

	紫外线特征	初始簇	黑色素瘤患者的存活时间	黑色素瘤导致的死亡
TCGA-BF-A1PU	紫外线特征	keratin	NaN	NaN
TCGA-BF-A1PV	紫外线特征	keratin	13.0	0.0
TCGA-BF-A1PX	紫外线特征	keratin	NaN	NaN
TCGA-BF-A1PZ	紫外线特征	keratin	NaN	NaN
TCGA-BF-A1Q0	非紫外线特征	immune	17.0	0.0

我们拥有每位患者（行）的以下信息。

- **紫外线特征**
  紫外线可能会引发某种基因突变。通过检查突变特征，研究者可以推断出紫外线是否可能引起导致这些患者罹患癌症的突变。
- **初始簇**
  论文中用基因表达数据将患者分成了多个簇。这些簇按照簇中典型的基因类型进行分类。主要的簇有“immune”（$n = 168$；51%）、“keratin”（$n = 102$；31%）和“MITF-low”（$n = 59$；18%）。
- **黑色素瘤患者的存活时间**
  患者存活的天数。
- **黑色素瘤导致的死亡**
  如果患者因黑色素瘤死亡，则值为 1；如果患者依然存活或因其他原因死亡，则值为 0。

接下来我们要为聚类生成的每个患者组绘制一条**存活曲线**。这些曲线表示经过一段时间后仍然存活的患者比例。注意，有些数据是**右删失的**（right-censored），也就是说，在某些情况下，我们不知道患者的确切死亡时间，或者患者可能因与黑色素瘤无关的原因死亡。在存活曲线的时间段内，我们认为这些患者仍然是“存活的”，但更加精密的分析应该尽量估计出可能的死亡时间。

为了根据存活时间得出存活曲线，我们需要先创建一个步长函数，每次减少 $1/n$，这里的 n 是组中患者的数量。然后将这个函数与无删失存活时间进行匹配。

```
def survival_distribution_function(lifetimes, right_censored=None):
    """Return the survival distribution function of a set of lifetimes.

    Parameters
    ----------
    lifetimes : array of float or int
    The observed lifetimes of a population. These must be non-negative.
right_censored : array of bool, same shape as `lifetimes`
    A value of `True` here indicates that this lifetime was not observed.
```

```
        Values of `np.nan` in `lifetimes` are also considered to be
        right-censored.

    Returns
    -------
    sorted_lifetimes : array of float
        The
    sdf : array of float
        Values starting at 1 and progressively decreasing, one level
        for each observation in `lifetimes`.

    Examples
    --------

    In this example, of a population of four, two die at time 1, a
    third dies at time 2, and a final individual dies at an unknown
    time. (Hence, ``np.nan``.)

    >>> lifetimes = np.array([2, 1, 1, np.nan])
    >>> survival_distribution_function(lifetimes)
    (array([ 0.,  1.,  1.,  2.]), array([ 1.  ,  0.75,  0.5 ,  0.25]))
    """
    n_obs = len(lifetimes)
    rc = np.isnan(lifetimes)
    if right_censored is not None:
        rc |= right_censored
    observed = lifetimes[~rc]
    xs = np.concatenate( ([0], np.sort(observed)) )
    ys = np.linspace(1, 0, n_obs + 1)
    ys = ys[:len(xs)]
    return xs, ys
```

既然可以轻松地根据存活数据得出存活曲线，那么就可以绘制出曲线。我们再编写一个函数，按照簇标识对存活时间分组，并用不同的折线绘制出每个组。

```
def plot_cluster_survival_curves(clusters, sample_names, patients,
                                 censor=True):
    """Plot the survival data from a set of sample clusters.

    Parameters
    ----------
    clusters : array of int or categorical pd.Series
        The cluster identity of each sample, encoded as a simple int
        or as a pandas categorical variable.
    sample_names : list of string
        The name corresponding to each sample. Must be the same length
        as `clusters`.
    patients : pandas.DataFrame
        The DataFrame containing survival information for each patient.
        The indices of this DataFrame must correspond to the
        `sample_names`. Samples not represented in this list will be
        ignored.
    censor : bool, optional
        If `True`, use `patients['melanoma-dead']` to right-censor the
        survival data.
```

```
    """
    fig, ax = plt.subplots()
    if type(clusters) == np.ndarray:
        cluster_ids = np.unique(clusters)
        cluster_names = ['cluster {}'.format(i) for i in cluster_ids]
    elif type(clusters) == pd.Series:
        cluster_ids = clusters.cat.categories
        cluster_names = list(cluster_ids)
    n_clusters = len(cluster_ids)
    for c in cluster_ids:
        clust_samples = np.flatnonzero(clusters == c)
        # 去除不在存活数据中的患者
        clust_samples = [sample_names[i] for i in clust_samples
                         if sample_names[i] in patients.index]
        patient_cluster = patients.loc[clust_samples]
        survival_times = patient_cluster['melanoma-survival-time'].values
        if censor:
            censored = ~patient_cluster['melanoma-dead'].values.astype(bool)
        else:
            censored = None
        stimes, sfracs = survival_distribution_function(survival_times,
                                                         censored)
        ax.plot(stimes / 365, sfracs)

    ax.set_xlabel('survival time (years)')
    ax.set_ylabel('fraction alive')
    ax.legend(cluster_names)
```

现在我们可以用 fcluster 函数得到样本（计数数据中的列）的簇标识，并分别绘制每条存活曲线。fcluster 函数接受一个联系矩阵（如 linkage 函数的返回值）和一个阈值，并返回簇标识。很难**事先确定**这个阈值应该多大，但通过检查联系矩阵中的距离，我们可以为固定数量的簇选择一个合适的阈值。

```
from scipy.cluster.hierarchy import fcluster
n_clusters = 3
threshold_distance = (yc[-n_clusters, 2] + yc[-n_clusters+1, 2]) / 2
clusters = fcluster(yc, threshold_distance, 'distance')

plot_cluster_survival_curves(clusters, data_table.columns, patients)
```

基因表达档案的聚类似乎识别出了一种黑色素瘤的高度危险的子类型（簇 2），如图 2-2 所示。TCGA 的后续研究通过更强有力的聚类技术和统计检验为这项发现提供了支持。这只是关于黑色素瘤的最新研究，其他研究还识别出了白血病（血癌）、肠癌等更多癌症的子类型。虽然上述聚类技术非常不稳定，但还有很多其他更加强大的方法可以用于研究这个数据集和其他类似的数据集。[1]

注 1：The Cancer Genome Atlas Network. Genomic classification of cutaneous melanoma [J]. Cell, 2015, 7(161): 1681-1696.

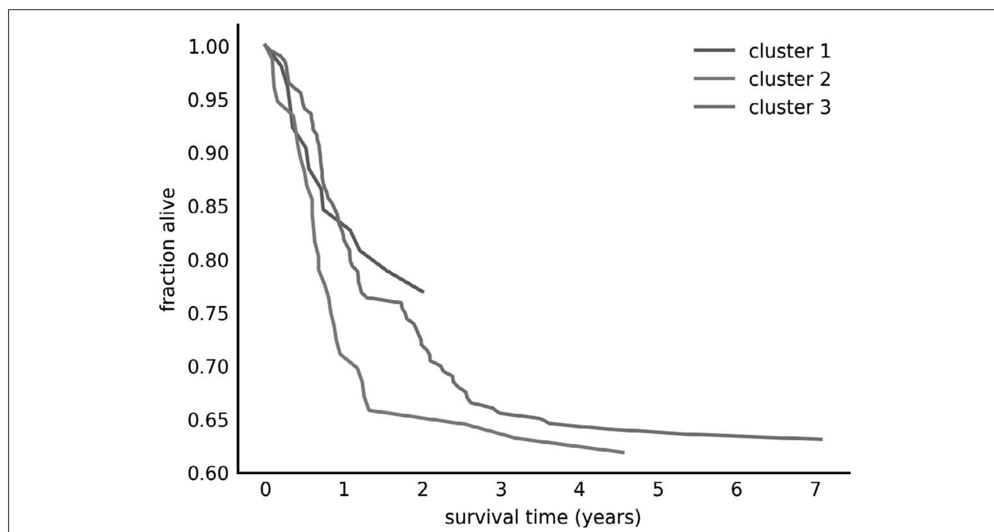


图 2-2：用基因表达数据得到的患者簇对应的存活曲线

2.5.1 进一步工作：使用TCGA患者簇

我们的聚类结果在预测存活率时会比论文中的簇表现得更好吗？使用紫外线特征会怎么样？分别用患者数据中的初始簇列和紫外线特征列绘制存活曲线。与我们的簇相比，它们的效果如何？

2.5.2 进一步工作：重新生成TCGA簇

我们留给你的练习是，实现这篇论文[2]中描述的方法。

(1) 对聚类样本的基因使用自助抽样（有放回随机抽样）。

(2) 为每个样本生成一个层次聚类。

(3) 在一个形状为 `(n_samples, n_samples)` 的矩阵中，保存一个样本对一起出现在自助聚类中的次数。

(4) 对结果矩阵执行层次聚类。

这样可以识别出频繁在聚类中同时出现的样本组，不受基因选择的影响。因此，可以认为这些样本是健壮聚集的簇。



提示

用 `np.random.choice` 和 `replacement=True` 创建行标号的自助抽样。

注 2：The Cancer Genome Atlas Network. Genomic classification of cutaneous melanoma [J]. Cell, 2015, 7(161): 1681-1696.

第 3 章

用ndimage实现图像区域网络

> 老虎！老虎！黑夜的森林中
> 燃烧着的煌煌的火光，
> 是怎样的神手或天眼
> 造出了你这样的威武堂堂？
>
> ——威廉·布莱克，《虎》

你可能知道数字图像是由**像素**组成的。一般来说，不应该将像素看作小方块，而应该将其看作**在规则网格上测量的**光信号的**点采样**。[1]

进一步讲，在处理图像时，我们要处理的对象经常比单个像素大得多。在一幅风景图像中，天空、土地、树木和岩石都包括多个像素。用于表示这种图像的常见数据结构称为区域邻接图（RAG，region adjacency graph）。其**节点**保存了图像每个区域的属性，其**链接**则保存了各个区域间的空间联系。在输入图像中，只要两个区域互相接触，那么它们所对应的两个节点间就有链接。

构建这样的数据结构是一项非常复杂的工作。若图像不是二维的，而是三维甚至四维的，那就更加困难了。这种情况在显微镜技术、材料科学和气候学等领域非常常见。但本章将向你展示如何使用 NetworkX（一个用于分析图像和网络的 Python 库）和一个来自 `ndimage`（SciPy 中的 *N* 维图像处理子模块）的过滤器，只用寥寥几行代码就能生成一个 RAG。

```
import networkx as nx
import numpy as np
from scipy import ndimage as ndi

def add_edge_filter(values, graph):
```

注 1：参见 Alvy Ray Smith 的技术笔记“A Pixel Is Not A Little Square”，1995 年 7 月 17 日。

```
    center = values[len(values) // 2]
    for neighbor in values:
        if neighbor != center and not graph.has_edge(center, neighbor):
            graph.add_edge(center, neighbor)
    return 0.0


def build_rag(labels, image):
    g = nx.Graph()
    footprint = ndi.generate_binary_structure(labels.ndim, connectivity=1)
    _ = ndi.generic_filter(labels, add_edge_filter, footprint=footprint,
                           mode='nearest', extra_arguments=(g,))
    return g
```

本书的由来

（来自胡安的笔记。）

应该强调一下本章，因为正是本章激发了我们撰写此书的热情。这段代码是 Vighnesh Birodkar 在本科阶段参加 2014 年的 Google 编程夏令营时编写的。当看到这段代码时，我真是佩服得五体投地。就本书而言，它涉及了 Python 科学计算的方方面面。学习完本章后，你应该不止能处理一维列表和二维表格，还能够完成任何维度的数组操作。除此之外，你还应该能理解图像过滤和网络处理的基本知识。

本章包括以下几项内容：将图像表示为 NumPy 数组、用 `scipy.ndimage` **过滤**图像，以及用 NetworkX 库在图形（网络）中建立图像区域。我们将逐次介绍这些内容。

3.1 图像就是NumPy数组

上一章中介绍过，NumPy 数组不但可以有效地表示表格数据，还能非常方便地执行各种计算。你将在本章中看到，数组在表示图像方面同样是行家里手。

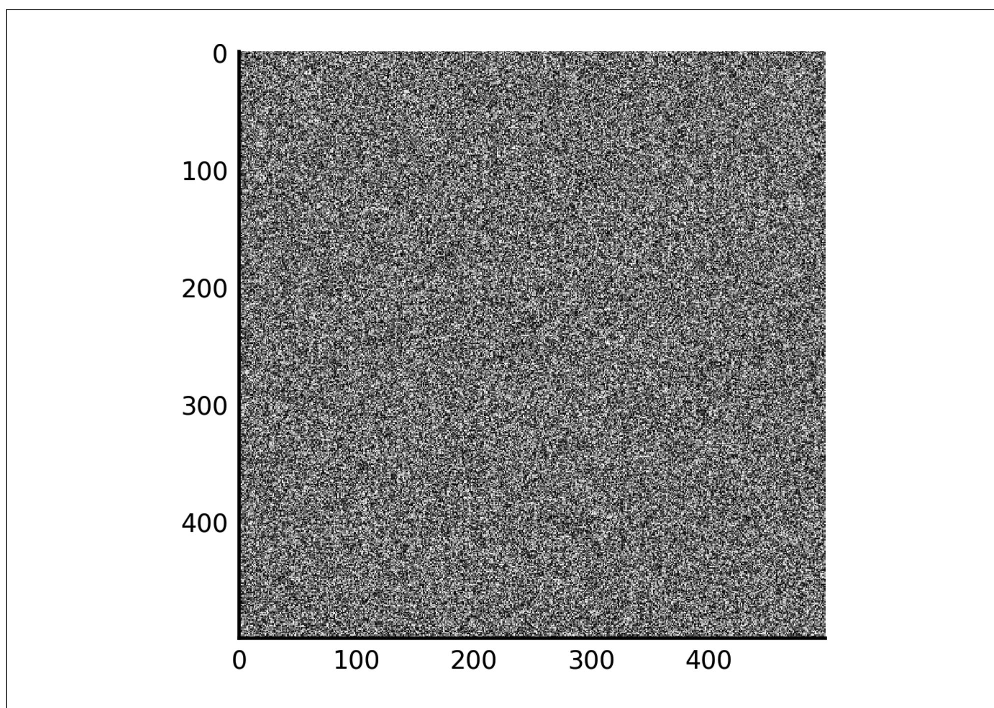
以下代码展示了如何仅用 NumPy 创建一幅白噪声图像，并用 Matplotlib 将其显示出来。首先，导入所需的包，然后用 IPython 神奇的 `matplotlib inline` 命令将图像显示在代码下面。

```
# 使图像显示在行中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

然后“制造一些噪声”，并将其显示为一幅图像。

```
import numpy as np
random_image = np.random.rand(500, 500)
plt.imshow(random_image);
```

`imshow` 函数可以将 NumPy 数组显示为一幅图像。



反之亦然：可以将一幅图像看作一个 NumPy 数组。在这种示例中，要使用 scikit-image 库，它是一个建立在 NumPy 和 SciPy 基础之上的图像处理工具集合。

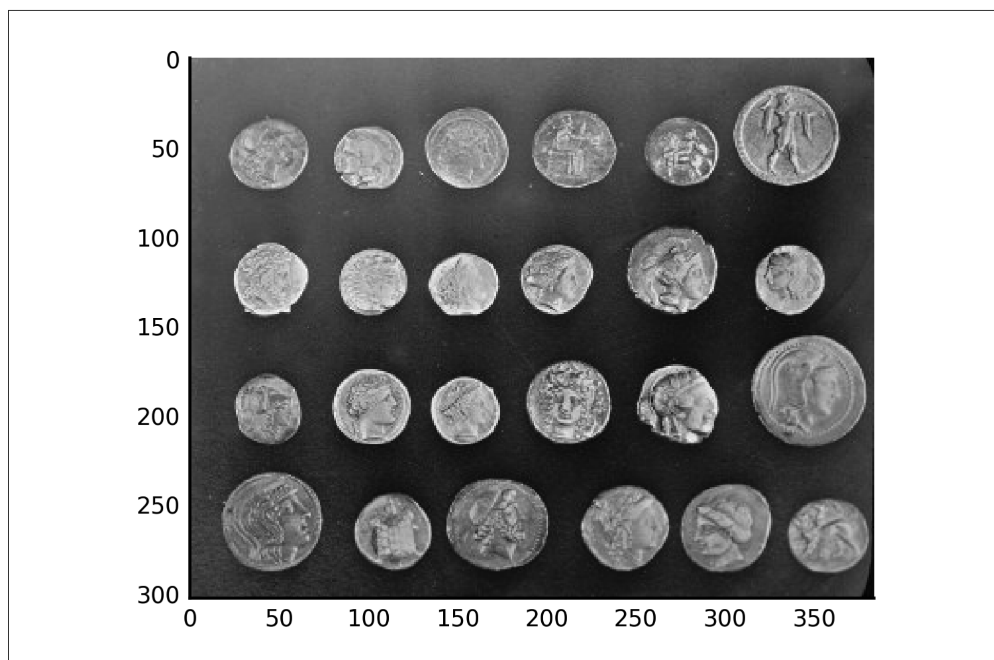
以下是一幅来自 scikit-image 库的 PNG 图像。这是一幅黑白图像（有时称为“灰度”），其中是一些收藏在布鲁克林博物馆的庞贝城的古罗马钱币。



下面是通过 scikit-image 加载的钱币图像。

```
from skimage import io
url_coins = ('https://raw.githubusercontent.com/scikit-image/scikit-image/'
             'v0.10.1/skimage/data/coins.png')
coins = io.imread(url_coins)
print("Type:", type(coins), "Shape:", coins.shape, "Data type:", coins.dtype)
plt.imshow(coins);

Type: <class 'numpy.ndarray'> Shape: (303, 384) Data type: uint8
```

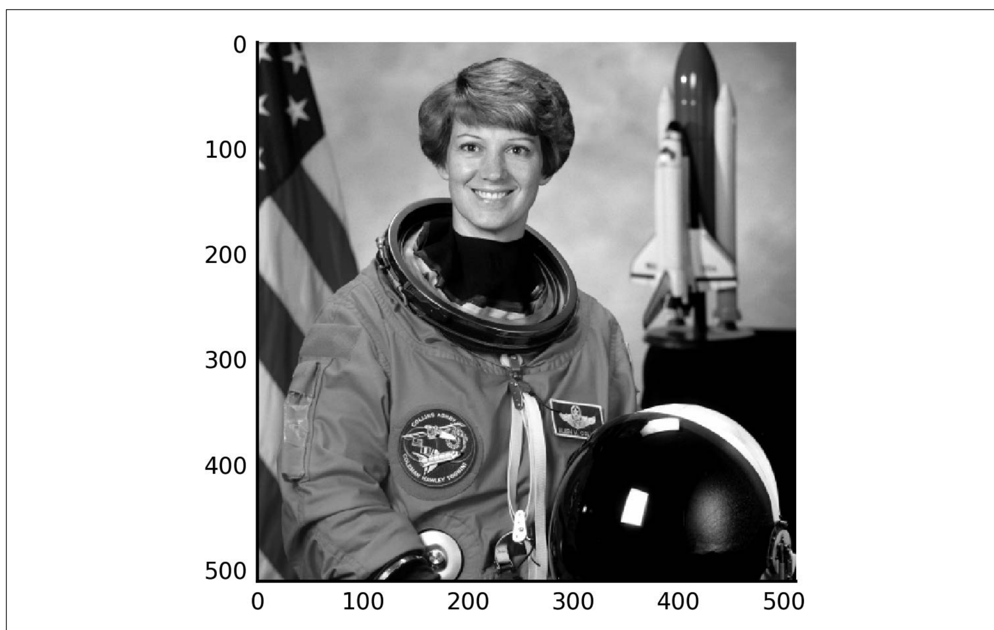


灰度图像可以表示为二维数组，每个数组元素包含相应位置上的灰度强度。因此，**图像就是 NumPy 数组**。

彩色图像是三维数组，其中前两个维度表示图像的空间位置，最后一个维度表示颜色通道，典型的就是红、绿、蓝这三种可以相加的基本颜色。为了展示如何使用这些维度，我们用宇航员艾琳·柯林斯的照片做一下示范。

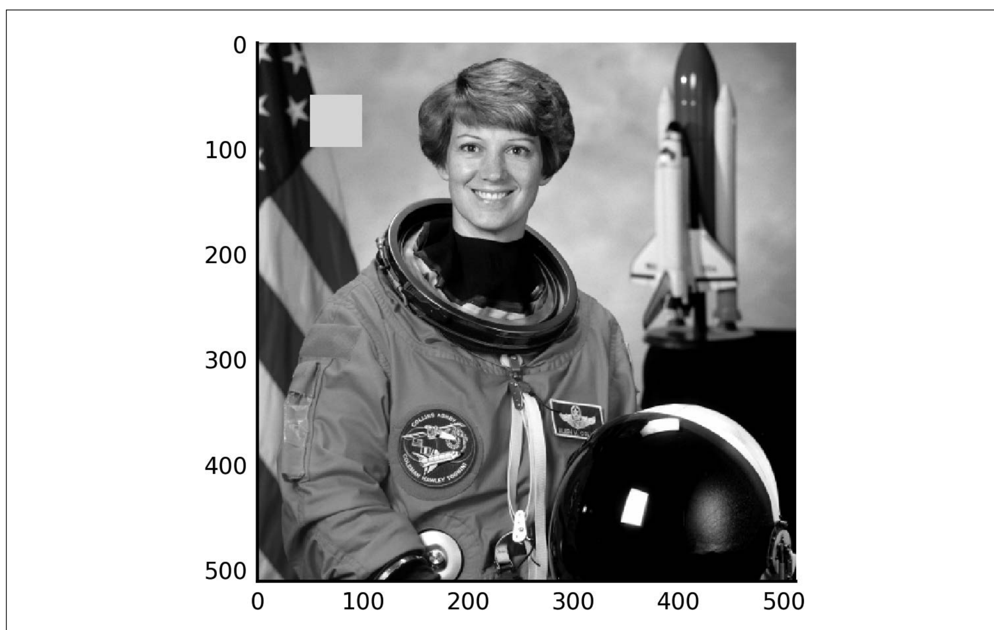
```
url_astronaut = ('https://raw.githubusercontent.com/scikit-image/scikit-image/'
                 'master/skimage/data/astronaut.png')
astro = io.imread(url_astronaut)
print("Type:", type(astro), "Shape:", astro.shape, "Data type:", astro.dtype)
plt.imshow(astro);

Type: <class 'numpy.ndarray'> Shape: (512, 512, 3) Data type: uint8
```



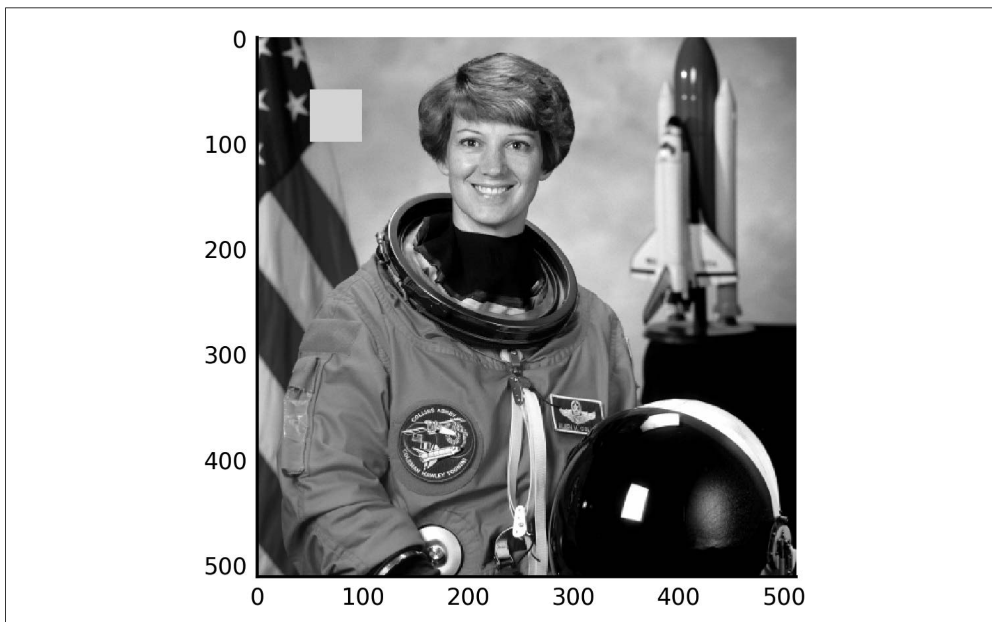
这幅图像就是 NumPy 数组。搞清楚这一点后，可以用简单的 NumPy 切片操作非常轻松地为图像加上一个绿色方块。

```
astro_sq = np.copy(astro)
astro_sq[50:100, 50:100] = [0, 255, 0] # 红、绿、蓝
plt.imshow(astro_sq);
```



另一种方式是使用布尔型**遮罩**（mask），这是一个值为 True 或 False 的数组。第 2 章中使用过这种方式来选择表格中的行。在这个示例中，可以用与图像形状相同的数组来选择像素。

```
astro_sq = np.copy(astro)
sq_mask = np.zeros(astro.shape[:2], bool)
sq_mask[50:100, 50:100] = True
astro_sq[sq_mask] = [0, 255, 0]
plt.imshow(astro_sq);
```



练习：为图像覆盖一个网格

我们刚刚学会了如何选择一个方块区域并将其涂为绿色，你能将这个操作扩展到其他形状和颜色吗？创建一个函数，在彩色图像上画出一个蓝色网格，并将其应用到艾琳·柯林斯的照片上。你的函数应该接受两个参数：输入图像和网格间距。你可以用以下模板着手编写函数。

```
def overlay_grid(image, spacing=128):
    """Return an image with a grid overlay, using the provided spacing.

    Parameters
    ----------
    image : array, shape (M, N, 3)
        The input image.
    spacing : int
        The spacing between the grid lines.

    Returns
    -------
    image_gridded : array, shape (M, N, 3)
```

```
        The original image with a blue grid superimposed.
    """
    image_gridded = image.copy()
    pass # 用你的代码替换这一行……
    return image_gridded

# plt.imshow(overlay_grid(astro, 128)); # 去掉这行注释来测试你的函数
```

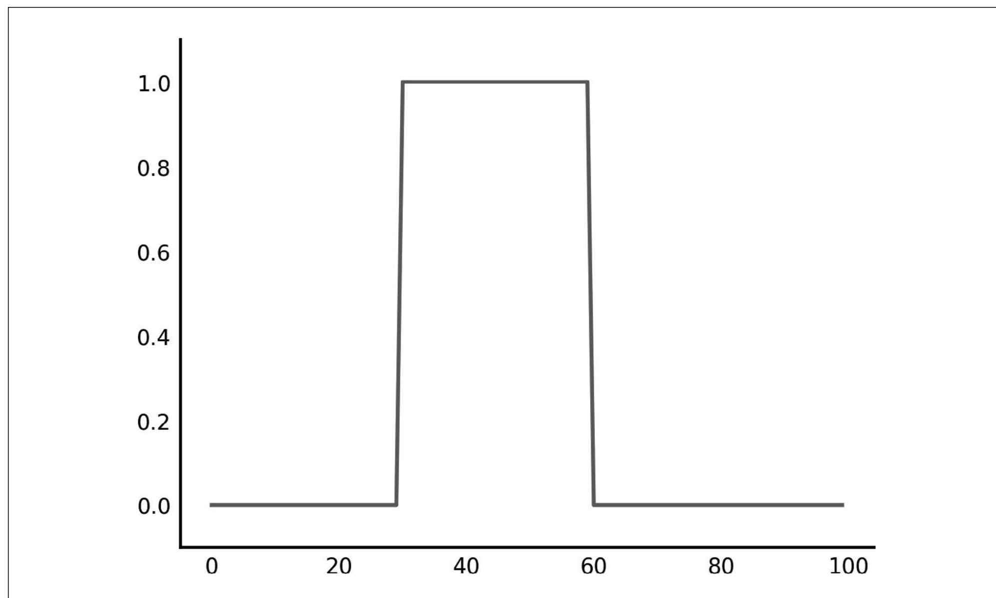
参见附录 A.1 节。

3.2 信号处理中的滤波器

滤波是图像处理中最基本和最常用的操作之一。你可以对图像进行滤波以去除噪声、增强特征或探测图像中的对象间的边缘。

理解滤波器最容易的方式不是图像，而是从一维信号开始。例如，你可能要测量到达光纤末端的光信号，如果每毫秒对信号进行一次**采样**，总测量时间是 100 毫秒，那么你可以得到一个长度为 100 的数组。假设开始测量 30 毫秒后打开光信号，持续 30 毫秒后关闭信号，那么你最后会得到如下的一个信号。

```
sig = np.zeros(100, np.float) #
sig[30:60] = 1 # 在30~60毫秒范围内，signal = 1，因为能够观测到光信号
fig, ax = plt.subplots()
ax.plot(sig);
ax.set_ylim(-0.1, 1.1);
```

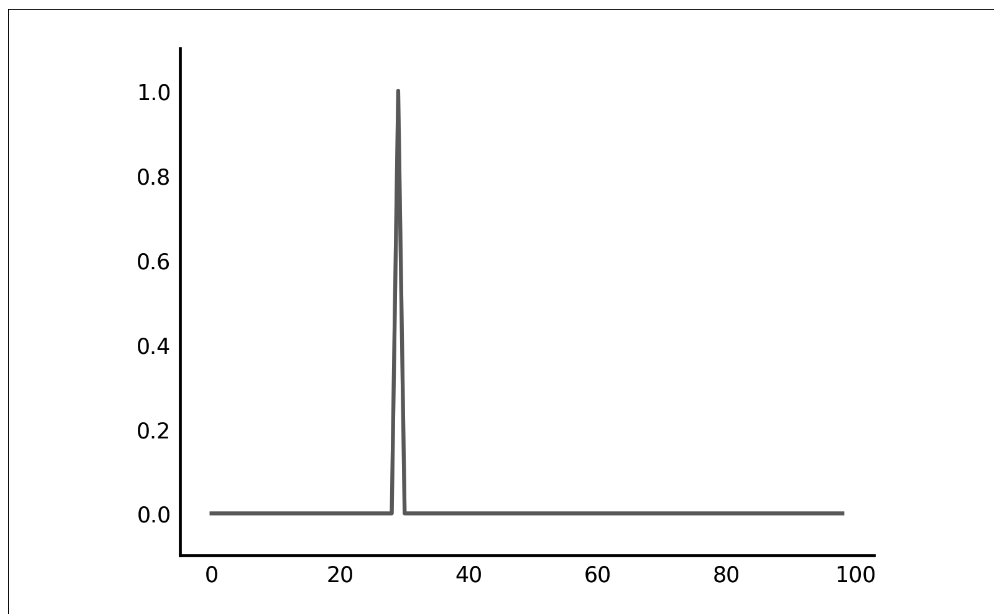


为了找出打开光信号的时间，你可以将光信号**延迟** 1 毫秒，然后从延迟信号中**减去**初始信号。这样一来，当信号从一个毫秒到下一个毫秒保持不变时，差为 0；当信号**增强**时，你就会得到一个正信号。

当信号**减弱**时，会得到一个负信号。如果只想精确地找出光信号的打开时间，那么可以对差分信号进行**裁剪**（clip）操作，这样任何负值都会转换为 0。

```
sigdelta = sig[1:] # sigdelta[0]等于sig[1]，以此类推
sigdiff = sigdelta - sig[:-1]
sigon = np.clip(sigdiff, 0, np.inf)
fig, ax = plt.subplots()
ax.plot(sigon)
ax.set_ylim(-0.1, 1.1)
print('Signal on at:', 1 + np.flatnonzero(sigon)[0], 'ms')

Signal on at: 30 ms
```



（这里使用了 NumPy 的 `flatnonzero` 函数，以得到 `sigon` 数组中第一个非零元素的索引。）

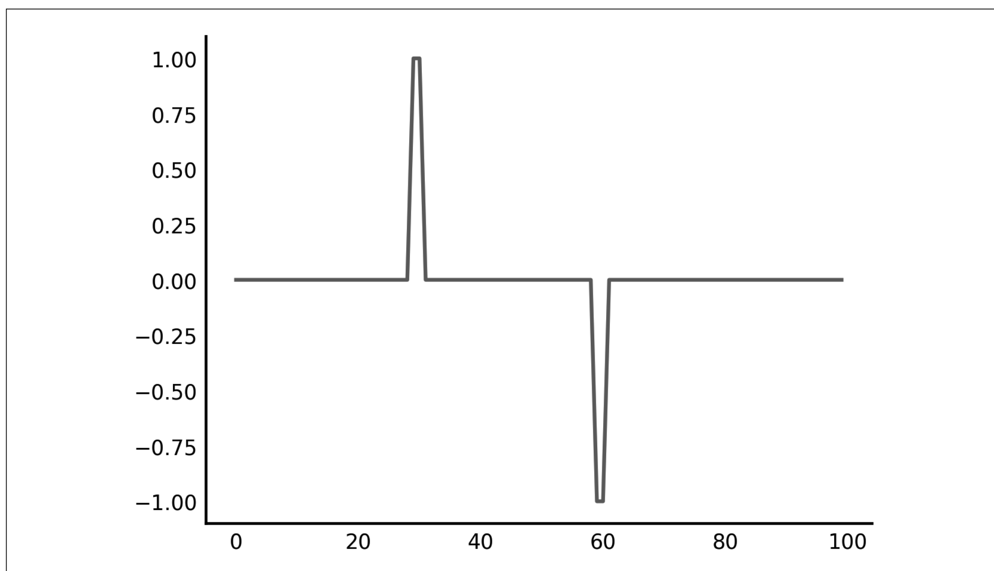
研究结果表明，可以通过一种名为**卷积**（convolution）的操作来完成滤波。在信号的所有点上，计算出围绕在该点附近的值与**核**（kernel，一个预设的值向量；或者**滤波器**）之间的点积，然后卷积操作会根据核的具体情况显示出信号的不同特征。

现在考虑一下核为差分滤波器 `(1, 0, -1)` 时信号 `s` 的情况。在任意位置 `i` 上，卷积结果是 `1*s[i+1] + 0*s[i] - 1*s[i-1]`，即 `s[i+1] - s[i-1]`。因此，当与 `s[i]` 邻接的值相同时，卷积结果为 0；当 `s[i+1] > s[i-1]`（信号增强）时，卷积结果为正；当 `s[i+1] < s[i-1]`（信号减弱）时，卷积结果为负。你可以将这个结果当作对输入函数的导数的估计。

一般情况下，卷积公式为 $s'(t) = \sum_{j=t-\tau}^{t} s(j) f(t-j)$。这里的 s 是信号，s' 是滤波后的信号，f 是滤波器，τ 是滤波器的长度。

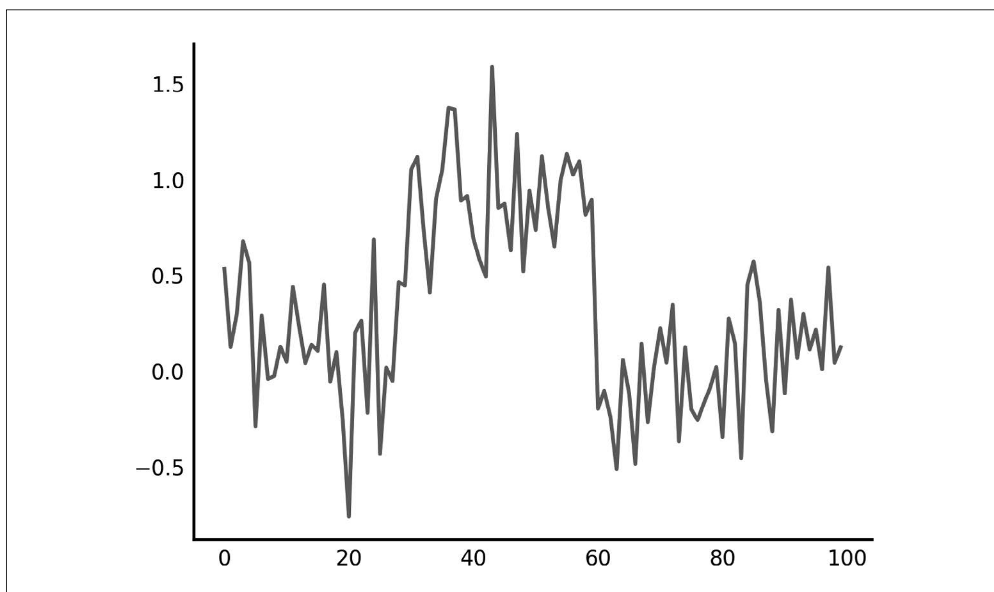
在 SciPy 中，可以用 `scipy.ndimage.convolve` 来进行卷积操作。

```
diff = np.array([1, 0, -1])
from scipy import ndimage as ndi
dsig = ndi.convolve(sig, diff)
plt.plot(dsig);
```



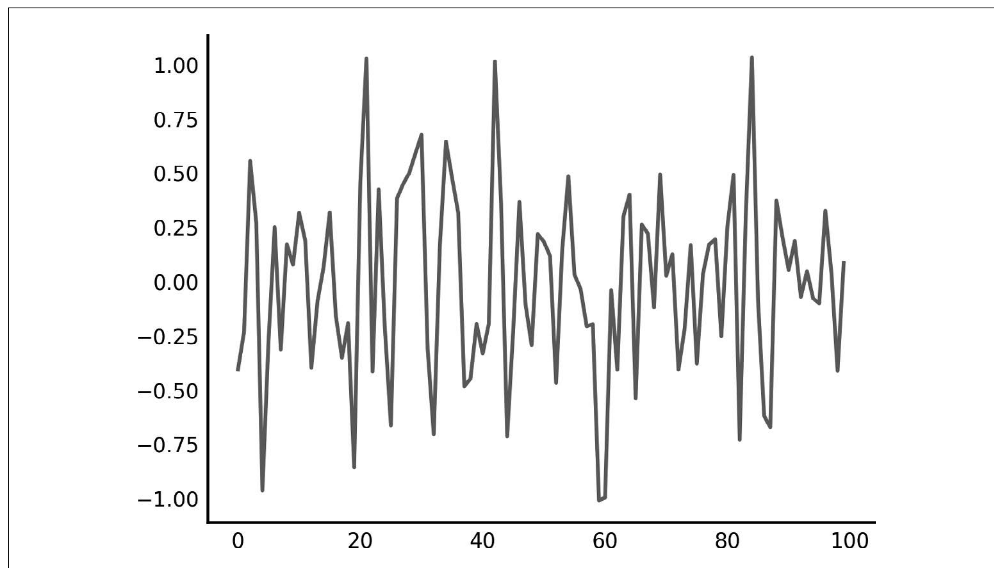
与前面相同，信号并不完美，经常是带有噪声的。

```
np.random.seed(0)
sig = sig + np.random.normal(0, 0.3, size=sig.shape)
plt.plot(sig);
```



普通的差分滤波器可以放大这些噪声。

```
plt.plot(ndi.convolve(sig, diff));
```



在这种情况下，可以向滤波器添加平滑的功能。最常用的平滑形式是**高斯平滑**，它用高斯函数对信号中相邻的点进行加权平均。可以编写一个函数来生成高斯平滑核，如下所示。

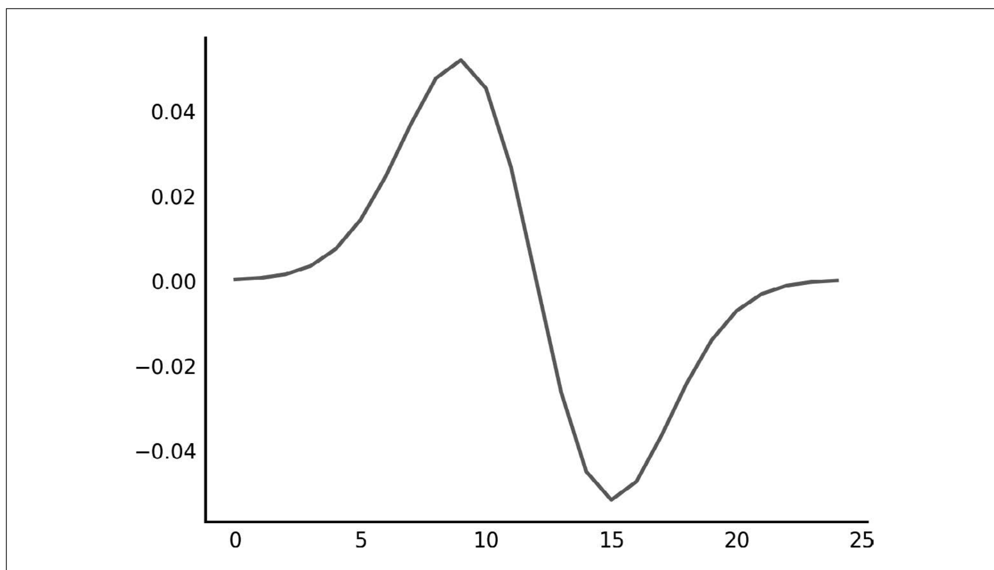
```
def gaussian_kernel(size, sigma):
    """Make a 1D Gaussian kernel of the specified size and standard deviation.

    The size should be an odd number and at least ~6 times greater than sigma
    to ensure sufficient coverage.
    """

    positions = np.arange(size) - size // 2
    kernel_raw = np.exp(-positions**2 / (2 * sigma**2))
    kernel_normalized = kernel_raw / np.sum(kernel_raw)
    return kernel_normalized
```

卷积的**结合性**（associative）是一个非常好的特性，这意味着，如果想要找出平滑后信号的导数，那么可以用平滑后的差分滤波器卷积信号！这样可以节省大量的计算时间，因为只需要对滤波器进行平滑，而滤波器一般远远小于信号数据。

```
smooth_diff = ndi.convolve(gaussian_kernel(25, 3), diff)
plt.plot(smooth_diff);
```



这个平滑差分滤波器在中心位置寻找边缘，并继续进行差分。在找到真正边缘而不是由噪声造成的“疑似”边缘的情况下，这个过程会持续发生。结果如图 3-1 所示。

```
sdsig = ndi.convolve(sig, smooth_diff)
plt.plot(sdsig);
```

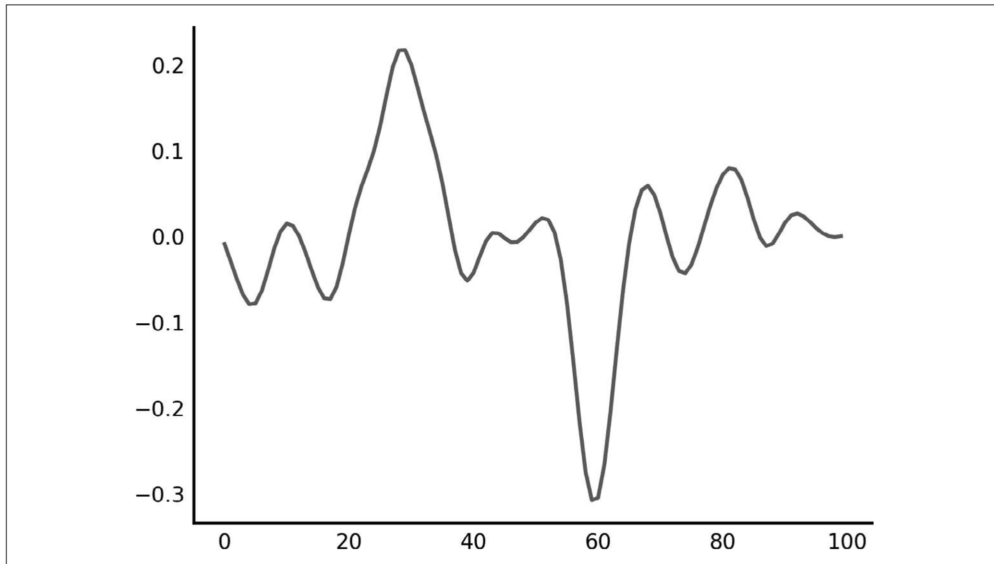


图 3-1：应用于噪声信号的平滑差分滤波器

虽然看上去还是此起彼伏，但这幅图中的**信噪比**（SNR，signal-to-noise ratio）已经比使用简单的差分滤波器好多了。



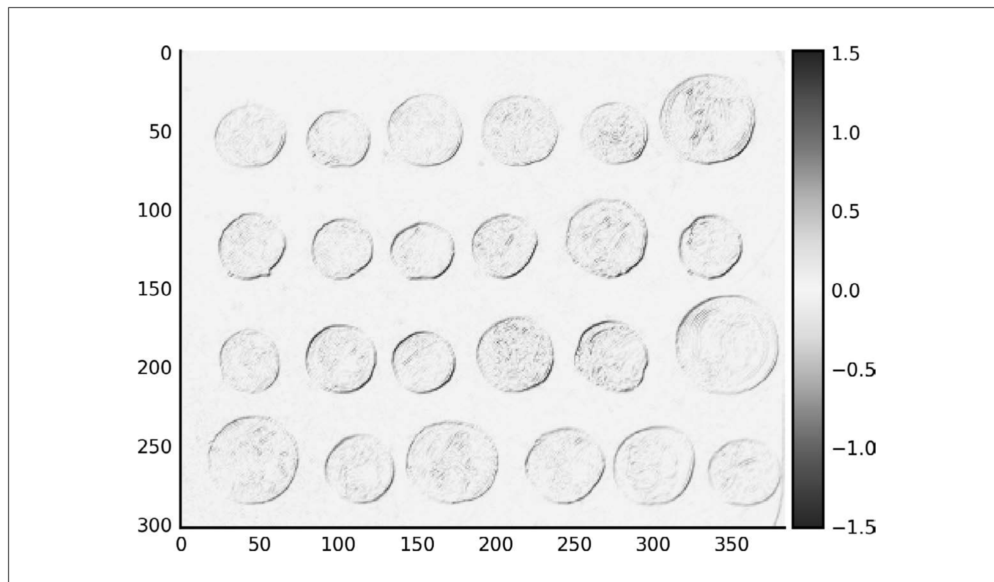
滤波

这种操作称为滤波的原因是，在物理电路中，很多这样的操作是由硬件实现的，这种硬件只允许某种特定的电流通过，而会阻止其他电流。这种硬件称为滤波器。例如，一种名为**低通**滤波器（low-pass filter）的常用滤波器可以从电流中去除高频电压波动。

3.3 图像滤波（二维滤波器）

我们已经了解了一维滤波，现在可以很自然地将这个概念扩展到二维信号，如图像。以下是一个可以在钱币图像中找出边缘的二维差分滤波器。

```
coins = coins.astype(float) / 255 # 防止溢出
diff2d = np.array([[0, 1, 0], [1, 0, -1], [0, -1, 0]])
coins_edges = ndi.convolve(coins, diff2d)
io.imshow(coins_edges);
```



二维滤波器与一维滤波器的原理相同：在图像的每个点上放置滤波器，计算滤波器值与图像值之间的点积，然后将结果放在输出图像中的相同位置。此外，与一维差分滤波器一样，当二维滤波器被放置在一个周围几乎没有差别的位置上时，点积会彼此抵消为 0。但当它被放置在图像亮度发生了改变的位置时，乘以 1 的值与乘以 −1 的值就不一样了，滤波后的输出会是一个正值或负值（取决于图像是否从右下角到左上角逐渐变亮）。

与一维滤波器一样，你可以通过合适的二维滤波器消除更加复杂和平滑的噪声。Sobel 滤波器就是设计用来做这种事的，它可以在水平和垂直两个变动方向上找出数据的边缘。先从水平滤波器开始。如果想要找出一张图片中的水平边缘，你可以试试下面的滤波器。

```
# 用（垂直的）列向量找出水平边缘
hdiff = np.array([[1], [0], [-1]])
```

然而，正如在一维滤波器中看到的那样，这会使图像中的边缘估计充满噪声。Sobel 滤波器不使用高斯平滑，因为那样会导致边缘模糊，它利用了图像中的边缘往往是连续的这一属性：例如，一幅海洋的图片包含的水平边缘应该是一条完整的线，而不是图像中几个特定的点。因此，Sobel 滤波器在水平方向上对垂直滤波器进行平滑：它在中心位置寻找一条被邻近位置支持的强边缘。

```
hsobel = np.array([[ 1,  2,  1],
                   [ 0,  0,  0],
                   [-1, -2, -1]])
```

垂直 Sobel 滤波器就是水平滤波器的简单转置。

```
vsobel = hsobel.T
```

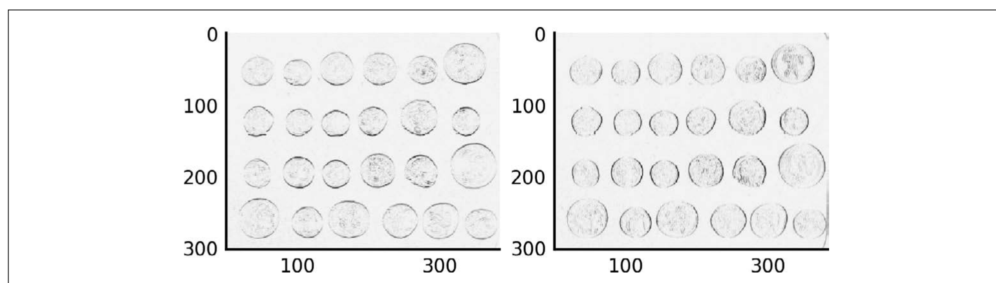
然后可以在钱币图像中找出水平边缘和垂直边缘。

```
# 定制x轴标签，使图形更易读
def reduce_xaxis_labels(ax, factor):
    """Show only every ith label to prevent crowding on x-axis,
        e.g., factor = 2 would plot every second x-axis label,
        starting at the first.

    Parameters
    ----------
    ax : matplotlib plot axis to be adjusted
    factor : int, factor to reduce the number of x-axis labels by
    """
    plt.setp(ax.xaxis.get_ticklabels(), visible=False)
    for label in ax.xaxis.get_ticklabels()[::factor]:
        label.set_visible(True)

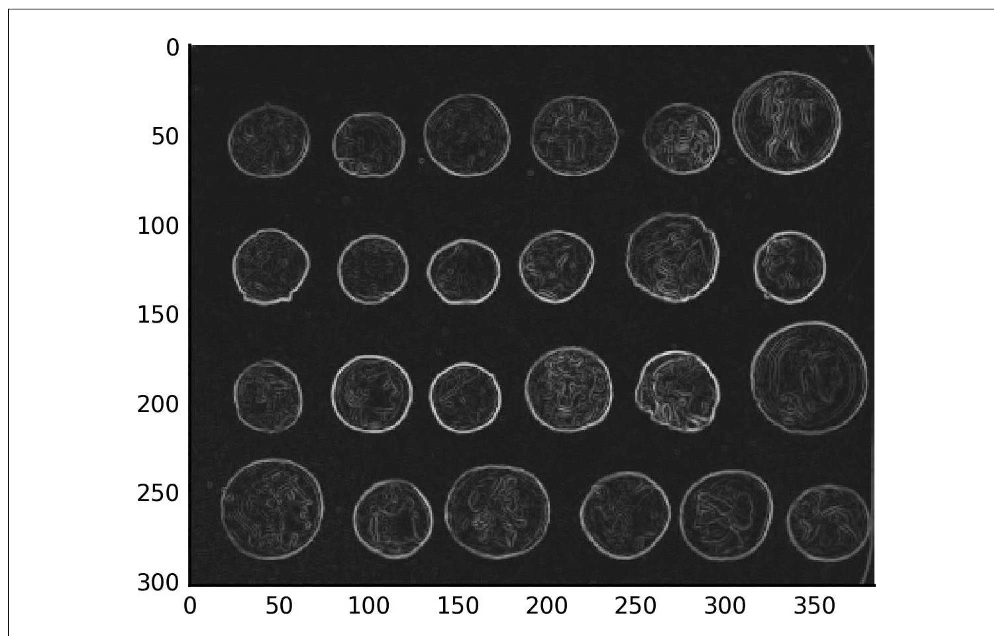

coins_h = ndi.convolve(coins, hsobel)
coins_v = ndi.convolve(coins, vsobel)

fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(coins_h, cmap=plt.cm.RdBu)
axes[1].imshow(coins_v, cmap=plt.cm.RdBu)
for ax in axes:
    reduce_xaxis_labels(ax, 2)
```



最后，就像毕达哥拉斯定理那样，你可以证明**任意**方向上的边缘大小等于平行分量和垂直分量的平方和的平方根。

```
coins_sobel = np.sqrt(coins_h**2 + coins_v**2)
plt.imshow(coins_sobel, cmap='viridis');
```

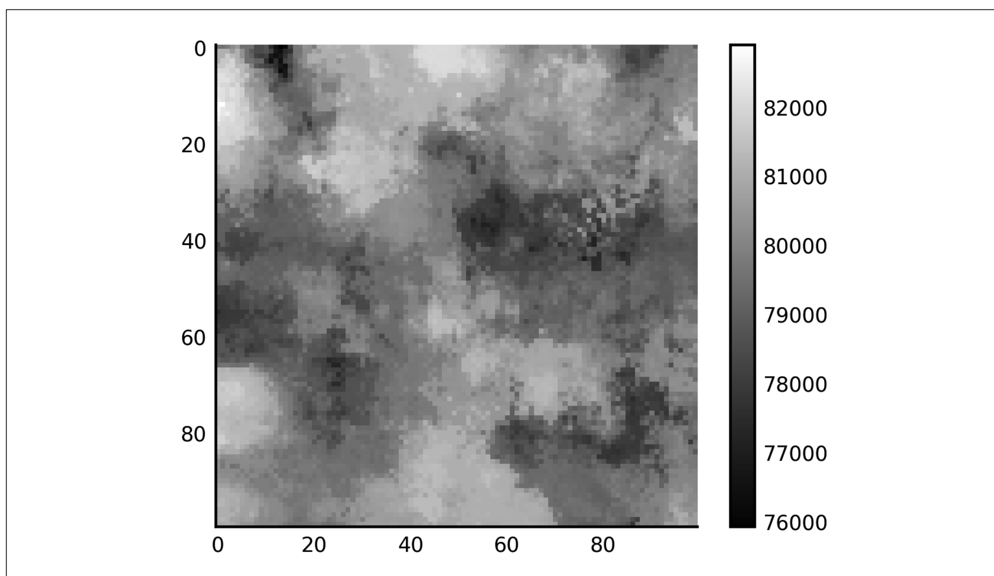


3.4 通用滤波器：邻近值的任意函数

除了通过 ndi.convolve 使用点积，SciPy 还允许你将滤波器定义为一个邻域内的点的**任意函数**，这是使用 ndi.generic_filter 实现的，这种方法可以让你表示出任意复杂的滤波器。

举个例子，假设我们想用一张图片表示某县的房屋价值中位数，粒度为 100 米 ×100 米。地方政府决定，房屋销售的计税方式是 10 000 美元加上方圆 1 千米内房价第 90 个百分位点的 5%。（因此，房价越高的地方，房屋销售成本也就越高。）可以用 generic_filter 生成一张能够显示所有地区税率的地图。

```
from skimage import morphology
def tax(prices):
    return 10000 + 0.05 * np.percentile(prices, 90)
house_price_map = (0.5 + np.random.rand(100, 100)) * 1e6
footprint = morphology.disk(radius=10)
tax_rate_map = ndi.generic_filter(house_price_map, tax, footprint=footprint)
plt.imshow(tax_rate_map)
plt.colorbar();
```



3.4.1 练习：康威的生命游戏

本节是根据 Nicolas Rougier 的建议添加的。

康威的生命游戏是一个看起来非常简单的游戏，游戏中有一个正方形网格，其中有很多“细胞”，这些细胞的生死取决于其周围的细胞。在每一个时间点，位置为 (i, j) 的细胞状态都要根据该细胞及其八位邻居（上、下、左、右以及四个角）的上一个状态来确定：

- 只有一个邻居或没有邻居的活细胞将死亡；
- 有两个或三个活着的邻居的活细胞可以存活到下一代；
- 有四个及以上活着的邻居的活细胞将因人口过多而死亡；
- 有三个活着的邻居的死细胞可以通过重生变为活细胞。

尽管这些规则看起来像一个人为设计的数学问题，但它们确实可以产生一些不可思议的模式。最简单的模式包括滑翔者（活细胞会逐代缓慢移动的小模式）或滑翔者枪（可以不断释放出滑翔者的固定模式），复杂一些的包括素数生成机（参见 Nathaniel Johnston 的“Generating Sequences of Primes in Conway’s Game of Life”），甚至能够模拟生命游戏本身！

你可以用 `ndi.generic_filter` 实现生命游戏吗？

参见附录 A.2 节。

3.4.2 练习：Sobel梯度幅值

前面介绍了如何将两个不同滤波器的输出组合起来，如 Sobel 水平滤波器和垂直滤波器。你能够编写一个函数，使用 `ndi.generic_filter` 通过一次传递完成这个操作吗？

参见附录 A.3 节。

3.5 图与NetworkX库

图是对错综复杂的数据的一种自然表示。例如，网页可以表示为节点，而网页之间的链接则可以表示为节点间的链接。或者，在生物学中，所谓的**转录网络**（transcription network）用节点表示基因，并用边来连接那些在表达上互相有直接影响的基因。



图与网络

在这个上下文中，术语“图”（graph）与“网络”是同义词，与图表的“图”（plot）不是一回事。数学家和计算机科学家分别发明了略有不同的名词来描述这些概念：图 = 网络，顶点 = 节点，边 = 链接 = 弧。与大多数人一样，我们会交替地使用这些名词。

或许你更熟悉网络术语：网络由**节点**（node）和节点之间的**链接**（link）组成。与之对应的是，图由**顶点**（vertex）和顶点之间的**边**（edge）组成。在 NetworkX 中，`Graph` 对象由 `nodes` 和节点之间的 `edges` 组成，这可能是最常见的用法。

为了介绍图的概念，我们将重新生成 Lav Varshney 等人的论文“Structural Properties of the Caenorhabditis elegans Neuronal Network”中的一些结果。

在示例中，我们将线虫神经系统中的神经元表示为节点，当一个神经元与另一个神经元之间产生突触（synapse）时，就在这两个节点之间放一条边。（**突触**是神经元用来交流信息的化学连接。）这种线虫是进行神经连接分析的绝妙例子，因为它们都具有相同数目的神经元（302 个），而且所有神经元间的连接都是已知的。这衍生出了一个妙趣横生的开放线虫项目（OpenWorm project），建议你多了解一下相关信息。

你可以从 WormAtlas database（http://www.wormatlas.org/neuronalwiring.html#Connectivity-data）下载 Excel 格式的神经元数据集。因为 pandas 库可以直接从网络上读取 Excel 表格，所以用它来读取数据，然后再输入到 NetworkX 中。

```
import pandas as pd
connectome_url = 'http://www.wormatlas.org/images/NeuronConnect.xls'
conn = pd.read_excel(connectome_url)
```

现在 `conn` 中包含一个 pandas `DataFrame`，其行结构如下所示。

```
[Neuron1, Neuron2, connection type, strength]
```

因为只想研究化学突触间的连接情况，所以用以下代码过滤其他突触类型。

```
conn_edges = [(n1, n2, {'weight': s})
              for n1, n2, t, s in conn.itertuples(index=False, name=None)
              if t.startswith('S')]
```

（可以参考 WormAtlas 网页上对不同连接类型的描述。）我们在上面的字典中使用了 `weight`，因为它是 NetworkX 中表示边属性的一个特殊关键字。然后用 NetworkX 中的 `DiGraph` 类建立图对象。

```
import networkx as nx
wormbrain = nx.DiGraph()
wormbrain.add_edges_from(conn_edges)
```

现在可以研究这种网络的一些性质了。对于有向网络，研究者最关心的事情之一就是哪个节点对网络中的信息流最重要。具有高**中介中心性**（betweenness centrality）的节点是那些位于多对不同节点之间最短路径上的节点。思考一下铁路网，某些站点会连接多条路线，因此你必须在这些站点进行换乘，才能完成多个不同的旅行路线。这些站点就是具有高中介中心性的节点。

可以通过 NetworkX 轻松找出那些重要程度接近的神经元。在 NetworkX API 文档中的“centrality”条目下，`betweenness_centrality` 的文档字符串详细说明了这个函数接受一个图对象作为输入，并返回一个字典，将节点 ID 映射到中介中心性的值（浮点数）。

```
centrality = nx.betweenness_centrality(wormbrain)
```

这样就可以用 Python 的内置函数 `sorted` 找出具有最高中心性的神经元。

```
central = sorted(centrality, key=centrality.get, reverse=True)
print(central[:5])
```

```
['AVAR', 'AVAL', 'PVCR', 'PVT', 'PVCL']
```

返回的神经元是 AVAR、AVAL、PVCR、PVT 和 PVCL，它们都与线虫对刺激的反应有关：AVA 神经元将线虫的前感知接收器与负责后向运动的神经元相连，而 PVC 神经元则将后感知接收器与前向运动相连。

Varshney 等人研究了由总共 279 个神经元中的 237 个神经元组成的一个**强连通分量**（strongly connected component）的性质。在图中，**连通分量**是一组节点，其中每个节点都可以通过全体链接中的某条路径到达其他所有节点。这个连通分量是一个**有向图**，也就是说，边是从一个节点**指向**另一个结点的，而不仅是连接节点。在这种情况下，强连通分量中的所有结点都能够**以正确的方向**遍历链接，到达彼此。因此，$A \rightarrow B \rightarrow C$ 不是强连通的，因为没有从 B 或 C 到达 A 的路。$A \rightarrow B \rightarrow C \rightarrow A$ 则**是**强连通的。

在神经回路中，你可以将强连通分量看作回路中的“大脑”，每当处理发生时，其上游的节点可以作为输入，下游的节点可以作为输出。



神经网络中的环

循环神经回路的概念可以追溯到 20 世纪 50 年代。下面这段精彩文字摘自 Amanda Gefter 在科学杂志 *Nautilus* 上发表的文章“The Man Who Tried to Redeem the World with Logic”。

> 如果有人看见闪电划过夜空，眼睛就会向大脑发送信号，并在一个神经元链路中留下杂乱的痕迹。从这个链路中的任意给定神经元开始，可以追溯信号的传输步骤，并找出闪电发生的确切时间，除非这个链路是一个环。如果神经元链路是一个环，那么闪电的编码信息就会在这个环中无休止地打转，它与闪电实际发生的时间没有丝毫联系。正如 McCulloch 所说，它成了“从时间中挣脱而出的思想”，换句话说，就是一段记忆。

NetworkX 可以简单明了地从我们的 `wormbrain` 网络中找出最大的强连通分量。

```
sccs = nx.strongly_connected_component_subgraphs(wormbrain)
giantscc = max(sccs, key=len)
print(f'The largest strongly connected component has '
      f'{giantscc.number_of_nodes()} nodes, out of '
      f'{wormbrain.number_of_nodes()} total.')
```

```
The largest strongly connected component has 237 nodes, out of 279 total.
```

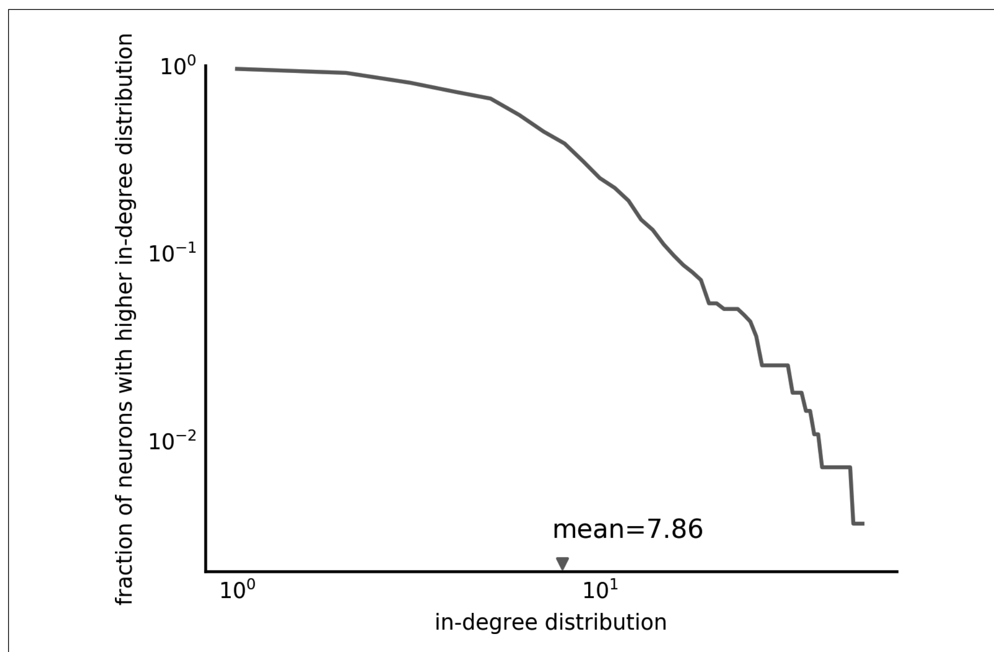
正如论文中提到的，这个分量碰巧比所预料的小，说明这个网络可以分离为输入层、中间层和输出层。

接下来重新生成论文中的图 6B，即入度分布的存活函数。首先，计算几个重要的数量。

```
in_degrees = list(wormbrain.in_degree().values())
in_deg_distrib = np.bincount(in_degrees)
avg_in_degree = np.mean(in_degrees)
cumfreq = np.cumsum(in_deg_distrib) / np.sum(in_deg_distrib)
survival = 1 - cumfreq
```

然后用 Matplotlib 绘图。

```
fig, ax = plt.subplots()
ax.loglog(np.arange(1, len(survival) + 1), survival)
ax.set_xlabel('in-degree distribution')
ax.set_ylabel('fraction of neurons with higher in-degree distribution')
ax.scatter(avg_in_degree, 0.0022, marker='v')
ax.text(avg_in_degree - 0.5, 0.003, 'mean=%.2f' % avg_in_degree)
ax.set_ylim(0.002, 1.0);
```



就是这样，用 SciPy 重新进行一次科学分析。没有进行曲线拟合……这正是以下练习要做的事情。

练习：用SciPy进行曲线拟合

这个练习有点像是对第 7 章（最优化）的热身：`scipy.optimize.curve_fit` 函数用一个幂律分布来拟合入度存活函数的尾部，$f(d) \sim d^{-\gamma}$，$d > d_0$，$d_- = 10$，$d_0 = 10$（即论文图 6B 中的红线），并修改绘图，使其包含那条红线。

参见附录 A.4 节。

现在你应该基本理解，图是一种抽象的科学概念，并知道如何用 Python 和 NetworkX 轻松实现对图的操作与分析。接下来介绍一种用于图像处理和计算机视觉的特殊类型的图。

3.6 区域邻接图

RAG 是图的一种表示方式，它易于进行**图像分割**：将图像划分为有意义的区域（**分段**）。如果你看过《终结者 2》，那么就应该见过图像分割（见图 3-2）。



图 3-2：终结者视角

图像分割对于人类来说不值一提，人类每时每刻都在进行图像分割，根本不用思考。但对计算机来说，图像分割则是一个非常困难的问题。为了理解这种困难，查看以下图形。



虽然你看到的是一张脸，但计算机只能看到一堆数字。

```
58688888888888899998898988888666532121
66888886888998999999899998888888865421
66665566566689999999999998888888888653
66668899998655688999899988888668665554
66888899998888888889988888665666666543
```

```
66888888886868868888999888866668888865
66666443334556688889988866666666668866
66884235221446588889988665644644444666
86864486233664666889886655464321242345
86666658333685588888866655659381366324
88866686688666866888888665858842248543
88888888888868868888886656668666656544
88888888868666888888886655668866668655
88888988888888888888888665688868888666
88889999989998888888886666888888886888
88889998888888888888886566888888888886
88888998888888688888666566868868888888
68888999888888888886888665688888888886
68888999998888888868888865568888888888
68888999886686666888688865656688888888
88888888886668888888888865655888888888
68888886665668888888988855555568888888
86868868658668868688888655555558888686
66688866468866855566655445555656888886
66688654888886868666555554556666666686
88688658688888888888666665555668668866
68888886666688888898888888666666566866
66888888845686888999888886666655686665
66688888862456668866666654431268686665
68688898886689696666665565531366868865
68888898888668998998998885356888986665
68688889888866899999999866666668986665
68888888888866666888866666666668886665
56888888888686889986868655566688888655
36668888888868888868688666686668886665
26686888888888888888886666886888865654
28688888888888888888866866666868666655
28666688888888888886866866868888666554
```

我们的视觉系统高度优化了人脸识别，即便是在这堆数字中，你都可能看到人脸！希望你能理解我们的意思。你还可以看看 Faces in Things 的 Twitter，它以一种更加幽默的方式演示了我们视觉系统的人脸识别优化过程。

无论如何，最大的困难是如何找出这些数字背后的意义，以及如何确定将图像分割为不同部分的边界位置。一种常用的方法是，先找出那些你能**确定**属于同一部分的小型区域（称为超像素），然后按照某种更加复杂的规则将它们合并起来。

举一个来自伯克利图像分割数据集（BSDS，Berkeley segmentation dataset）的简单例子，假设你想将以下图像中的老虎分割出来。

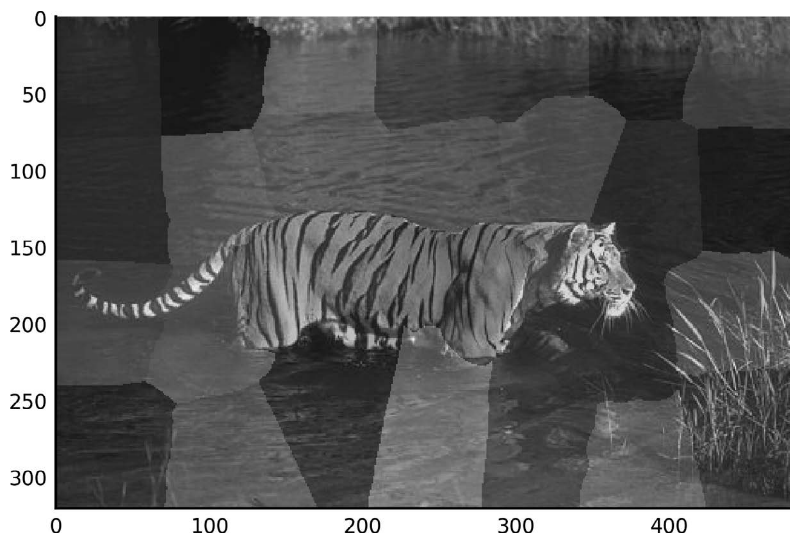


名为**简单线性迭代聚类**（SLIC，simple linear iterative clustering）的聚类算法可以提供一个不错的起点，scikit-image 库中提供了这种算法。

```
url = ('http://www.eecs.berkeley.edu/Research/Projects/CS/vision/'
       'bsds/BSDS300/html/images/plain/normal/color/108073.jpg')
tiger = io.imread(url)
from skimage import segmentation
seg = segmentation.slic(tiger, n_segments=30, compactness=40.0,
                        enforce_connectivity=True, sigma=3)
```

scikit-image 库中还有一个展示分割的函数，可用于对简单线性迭代聚类进行可视化。

```
from skimage import color
io.imshow(color.label2rgb(seg, tiger));
```

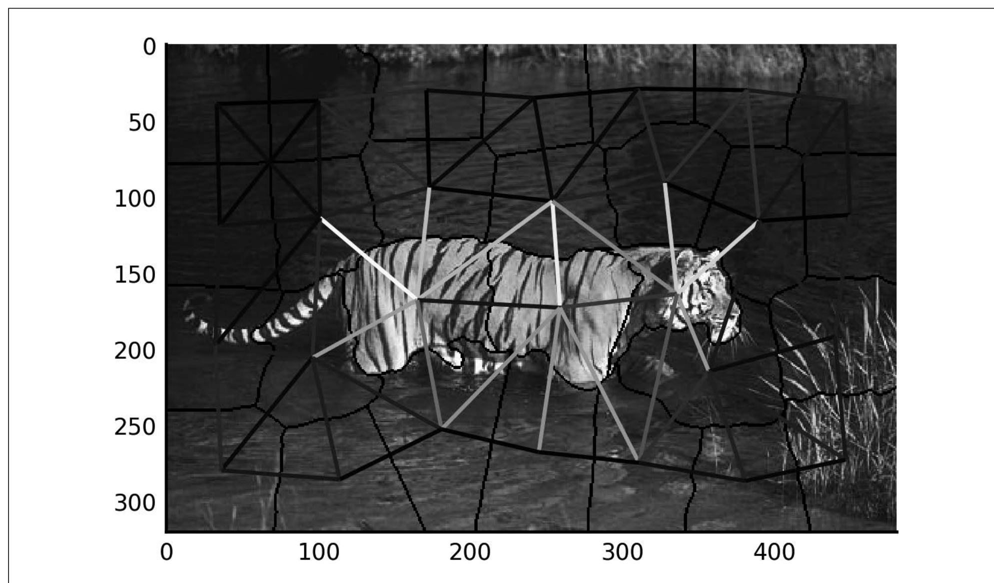


从图中可以看出，老虎的身体被分成了 3 个部分，图像的其余部分也进行了分割。

RAG 中的每个节点表示图像中的一块区域，当两块区域相邻时，就用一条边将两个节点连接起来。在实际建立一个 RAG 之前，可以先用 scikit-image 中的 `show_rag` 函数将上图中的 RAG 显示出来，以感觉一下其形式。实际上，本章的代码片段都来自 scikit-image 库！

```
from skimage.future import graph

g = graph.rag_mean_color(tiger, seg)
graph.show_rag(seg, g, tiger);
```



在这张图中，你可以看到与每个区域对应的节点，以及相邻区域之间的边。它们使用 Matplotlib 中的 YlGnBu（黄－绿－蓝）颜色图按照相邻两个节点颜色不同的原则进行着色。

上图还展示了将图像分割表示为图的奇妙之处：有些边是位于老虎的体内节点和体外节点之间的，相比于那些体内节点之间的边或体外节点之间的边，这些边更加明亮（意味着更高价值）。因此，沿着这些边切割图像就能得到想要的分割结果。我们选择的是一种相对简单的基于颜色的图像分割，但对于具有更复杂的成对关系的图来说，以上原则同样适用。

3.7 优雅的ndimage：如何根据图像区域建立图对象

万事俱备：现在你已经了解了 NumPy 数组、图像滤波、通用滤波器、图和 RAG。接下来编写一个程序，将老虎从图中抠出来！

显而易见的方法是用两个嵌套的 `for` 循环迭代图像中的每个像素，检查相邻像素并核对不同的标签。

```
import networkx as nx
def build_rag(labels, image):
    g = nx.Graph()
    nrows, ncols = labels.shape
    for row in range(nrows):
        for col in range(ncols):
            current_label = labels[row, col]
            if not current_label in g:
                g.add_node(current_label)
                g.node[current_label]['total color'] = np.zeros(3, dtype=np.float)
                g.node[current_label]['pixel count'] = 0
            if row < nrows - 1 and labels[row + 1, col] != current_label:
                g.add_edge(current_label, labels[row + 1, col])
            if col < ncols - 1 and labels[row, col + 1] != current_label:
                g.add_edge(current_label, labels[row, col + 1])
            g.node[current_label]['total color'] += image[row, col]
            g.node[current_label]['pixel count'] += 1
    return g
```

嘿！代码确实有效，但如果想要分割一个三维图像，你就不得不重写一下代码。

```
import networkx as nx
def build_rag_3d(labels, image):
    g = nx.Graph()
    nplns, nrows, ncols = labels.shape
    for pln in range(nplns):
        for row in range(nrows):
            for col in range(ncols):
                current_label = labels[pln, row, col]
                if not current_label in g:
                    g.add_node(current_label)
                    g.node[current_label]['total color'] = np.zeros(3, dtype=np.float)
                    g.node[current_label]['pixel count'] = 0
                if pln < nplns - 1 and labels[pln + 1, row, col] != current_label:
                    g.add_edge(current_label, labels[pln + 1, row, col])
                if row < nrows - 1 and labels[pln, row + 1, col] != current_label:
                    g.add_edge(current_label, labels[pln, row + 1, col])
                if col < ncols - 1 and labels[pln, row, col + 1] != current_label:
                    g.add_edge(current_label, labels[pln, row, col + 1])
                g.node[current_label]['total color'] += image[pln, row, col]
                g.node[current_label]['pixel count'] += 1
    return g
```

这两段代码都相当丑陋和笨重，而且难以扩展：如果想在相邻像素中考虑角上的像素（即 `[row, col]` 与 `[row + 1, col + 1]` 也是相邻像素），那么代码会变得更加凌乱。此外，如果想要分析三维视频，就必须添加另一个维度，再做一层循环嵌套。太乱了！

看看 Vighnesh 的天才做法：SciPy 的 `generic_filter` 函数已经替我们完成了迭代！前面用这个函数来计算关于 NumPy 数组每个元素邻域的任意复杂函数。现在我们不想通过这个函数得到滤波后的图像，而想要得到一个图。`generic_filter` 函数可以让你向滤波函数传递一个附加参数，可以用这个功能建立图对象。

```
import networkx as nx
import numpy as np
```

```
from scipy import ndimage as nd

def add_edge_filter(values, graph):
    center = values[len(values) // 2]
    for neighbor in values:
        if neighbor != center and not graph.has_edge(center, neighbor):
            graph.add_edge(center, neighbor)
    # 没有用到浮点型返回值，但generic_filter需要有这样的返回值。
    return 0.0

def build_rag(labels, image):
    g = nx.Graph()
    footprint = ndi.generate_binary_structure(labels.ndim, connectivity=1)
    _ = ndi.generic_filter(labels, add_edge_filter, footprint=footprint,
                           mode='nearest', extra_arguments=(g,))
    for n in g:
        g.node[n]['total color'] = np.zeros(3, np.double)
        g.node[n]['pixel count'] = 0
    for index in np.ndindex(labels.shape):
        n = labels[index]
        g.node[n]['total color'] += image[index]
        g.node[n]['pixel count'] += 1
    return g
```

这段代码真是熠熠生辉，原因如下。

- ndi.generic_filter 迭代数组元素**及其邻居**。（直接用 numpy.ndindex 在数组索引间迭代。）
- 我们从滤波函数中返回一个浮点数，因为 generic_filter 要求滤波函数返回浮点数。不过，我们会忽略滤波输出（所有输出都是 0），只用其向图中添加边的"副作用"。
- 循环没有好几层的嵌套，这使得代码更加紧凑，一气呵成。
- 代码适用于一维、二维、三维，甚至八维图像！
- 如果想要支持对角连接，只需要将 connectivity 参数修改为 ndi.generate_binary_structure。

3.8 归纳总结：平均颜色分割

接下来可以用学到的知识将老虎从上面的图像中分割出来。

```
g = build_rag(seg, tiger)
for n in g:
    node = g.node[n]
    node['mean'] = node['total color'] / node['pixel count']
for u, v in g.edges_iter():
    d = g.node[u]['mean'] - g.node[v]['mean']
    g[u][v]['weight'] = np.linalg.norm(d)
```

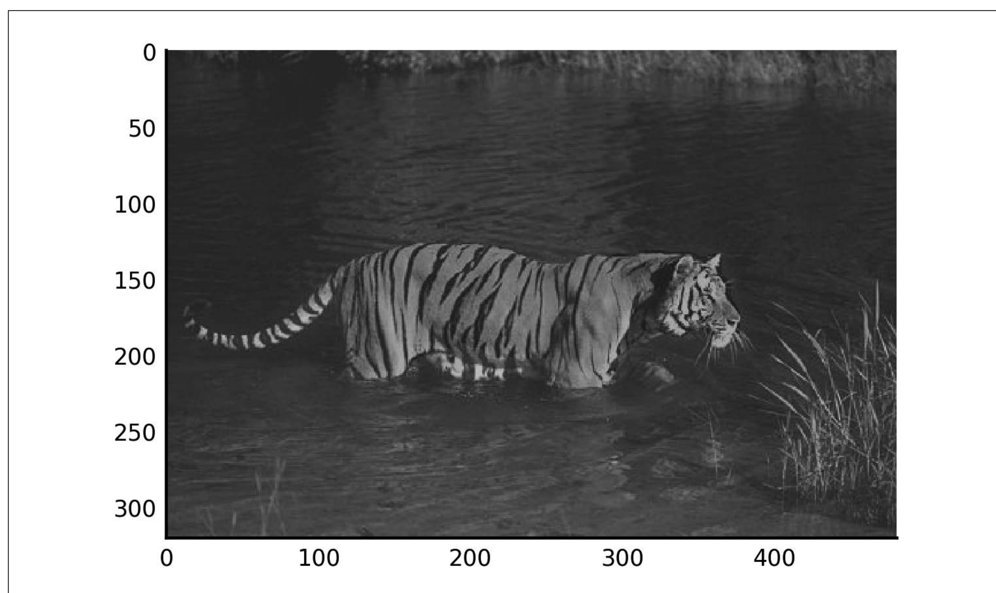
每条边都保存了它连接的两个区域的平均颜色的差。可以为图设定一个阈值：

```
def threshold_graph(g, t):
    to_remove = [(u, v) for (u, v, d) in g.edges(data=True)
                 if d['weight'] > t]
    g.remove_edges_from(to_remove)
threshold_graph(g, 80)
```

最后，体会一下我们在第 2 章中学到的 NumPy 用数组进行索引（即花式索引）的威力。

```
map_array = np.zeros(np.max(seg) + 1, int)
for i, segment in enumerate(nx.connected_components(g)):
    for initial in segment:
        map_array[int(initial)] = i
segmented = map_array[seg]
plt.imshow(color.label2rgb(segmented, tiger));
```

啊哦！老虎的尾巴似乎不见了！



尽管如此，我们仍然认为这是 RAG 能力的一次成功展示，同时也体现了 SciPy 和 NetworkX 实现它的美妙之处。这些功能很多都能在 scikit-image 库中找到。如果对图像分析感兴趣，就去查阅一下吧！

第 4 章

频率与快速傅里叶变换

如果想要了解宇宙的奥秘，那么就用能量、频率和振动来思考吧。

——尼古拉·特斯拉

本章是与斯特凡·范德瓦尔特的父亲——PW·范德瓦尔特——合作写成的。

本章的风格与其余部分有轻微差异，你会发现本章的**代码**非常朴实。本章将介绍一种优雅的**算法**——快速傅里叶变换（FFT，fast Fourier transform），其用途包罗万象。SciPy 实现了这种算法，当然，它也可以作用于 NumPy 数组。

4.1 频率的引入

我们从一些绘图风格的设置开始，并导入常用的库。

```
# 使图形显示在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')

import numpy as np
```

离散傅里叶变换（DFT，discrete Fourier transform）[1] 是一种用于将时间或空间数据转换为**频域**（frequency domain）数据的数学方法。**频率**是一个耳熟能详的概念，因为它经常出现在我们的日常用语中：使用耳机能听到的最低声音大概是 20 赫兹，而钢琴的中央 C 音的频率为 261.6 赫兹。赫兹是每秒振动次数，这里指的是耳机中的振动膜每秒来回移动的次数。这种振动在空气中产生压缩脉冲，并传送到你的鼓膜上，引起同样频率的振动。因此，如

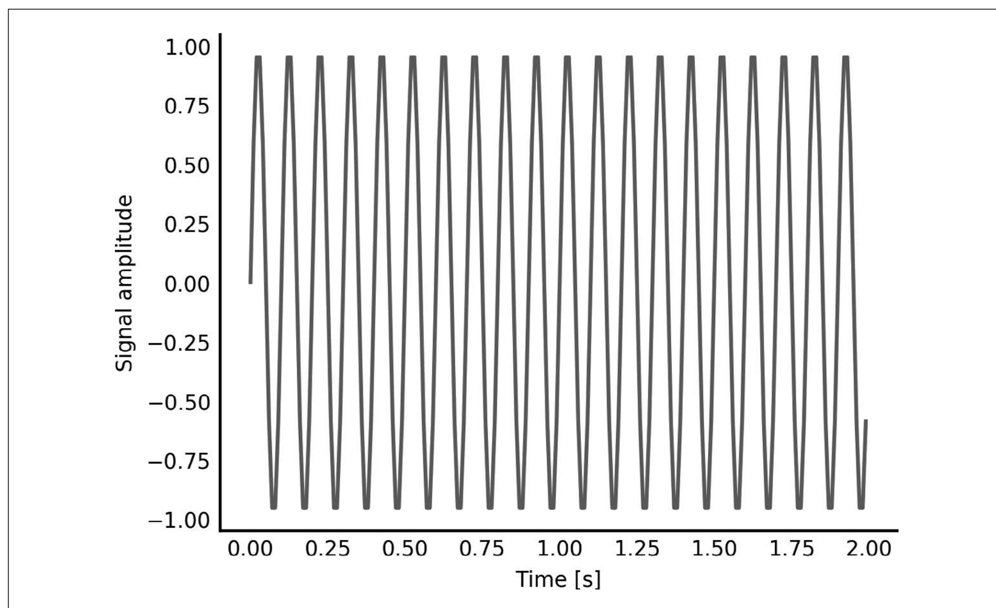
注 1：离散傅里叶变换处理的是采样数据，与之相对的标准傅里叶变换使用的是连续函数。

果你建立一个简单的周期函数，如 $\sin(10 \times 2\pi t)$，那么就可以将其看作一个波，如以下代码和图片所示。

```
f = 10 # 频率，以秒为周期，单位为赫兹
f_s = 100 # 采样率，每秒测量次数

t = np.linspace(0, 2, 2 * f_s, endpoint=False)
x = np.sin(f * 2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, x)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Signal amplitude');
```



或者也可以将它看作一个**频率**为 10 赫兹的重复信号（它每 1/10 秒重复一次，这个时间长度称为**周期**）。虽然我们会自然地将频率与时间联系在一起，但它同样可以用来描述空间。例如，一张纺织品印花的图片会呈现出高**空间频率**，而天空或其他平滑物体则具有低空间频率。

我们通过离散傅里叶变换的应用研究一下正弦波，如以下代码和图片所示。

```
from scipy import fftpack

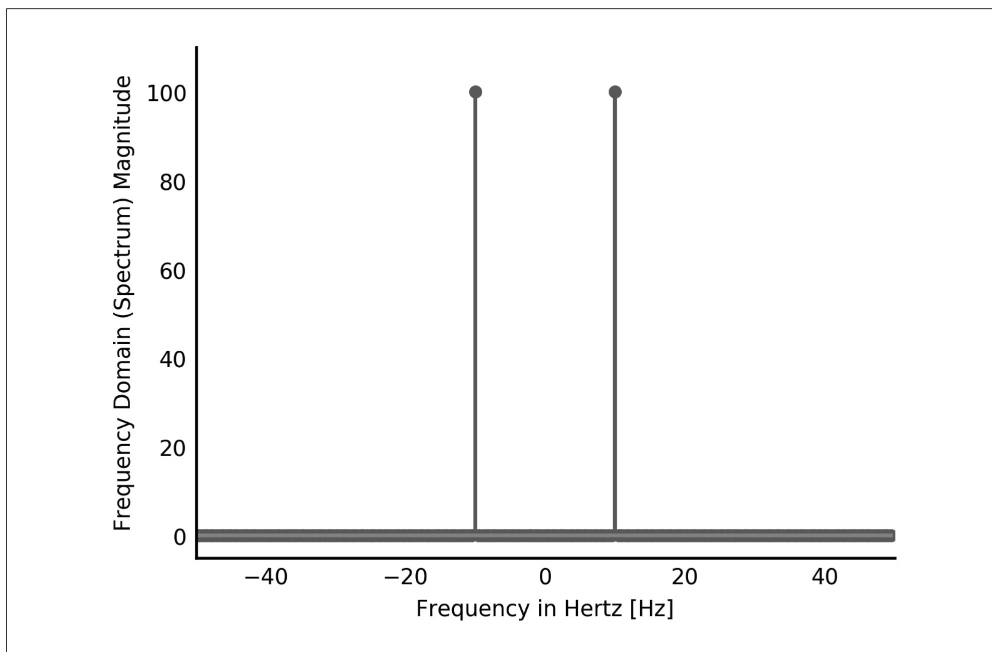
X = fftpack.fft(x)
freqs = fftpack.fftfreq(len(x)) * f_s

fig, ax = plt.subplots()

ax.stem(freqs, np.abs(X))
ax.set_xlabel('Frequency in Hertz [Hz]')
```

```
ax.set_ylabel('Frequency Domain (Spectrum) Magnitude')
ax.set_xlim(-f_s / 2, f_s / 2)
ax.set_ylim(-5, 110)
```

```
(-5, 110)
```



可以看到，快速傅里叶变换的输出是一个与输入形状相同的一维数组，其中包含的值为复数。除了两个频率项，其他所有值均为零。按照传统，用**茎叶图**对结果幅度进行可视化，其中每个茎的高度对应基础值。

（4.6.1 节的附注栏“离散傅里叶变换”将解释为什么会有正频率和负频率。你还可以通过这部分内容更加深入地了解傅里叶变换底层的数学知识。）

傅里叶变换可以带我们从**时域**转换到**频域**，它有着极其广泛的应用。**快速傅里叶变换**是一种计算离散傅里叶变换的方法，它可以在计算过程中保存之前的结果并重用，从而达到非常高的速度。

本章将研究离散傅里叶变换的几种应用，以说明快速傅里叶变换可以应用于多维数据（不仅限于一维测量），从而实现各种各样的目标。

4.2　示例：鸟鸣声谱图

我们从一个最普通的应用开始，将（随时间变化的空气压力组成的）声音信号转换为**声谱图**（spectrogram）。你可能在音乐播放器的均衡器视图或老式的立体声音响上见过声谱图（见图 4-1）。

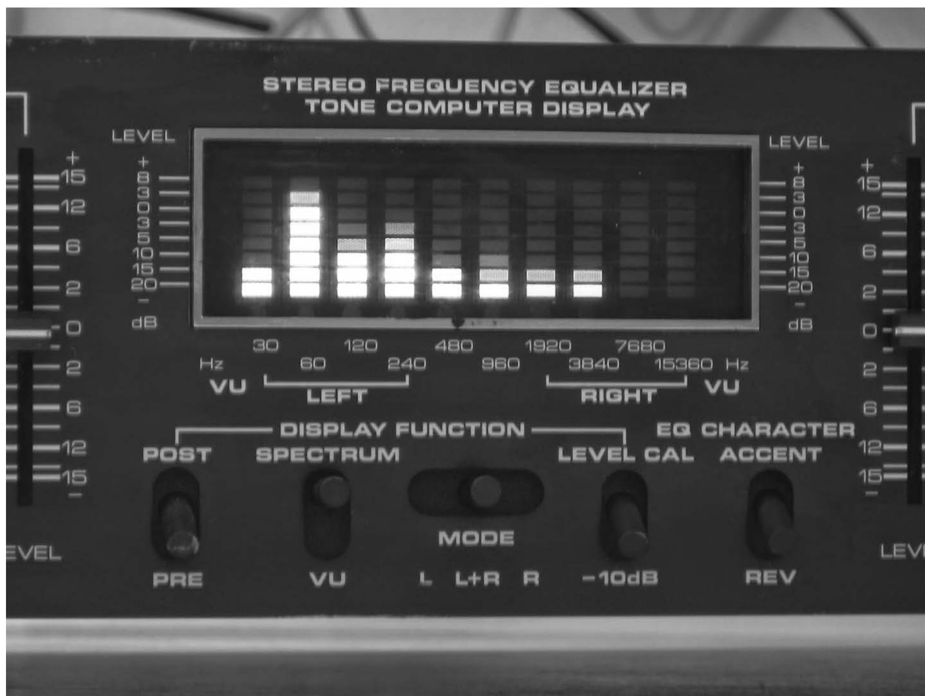


图 4-1：Numark EQ2600 立体声均衡器（图片获得了作者 Sergey Gerasimuk 的授权）

我们来欣赏一段夜莺的鸣叫（根据 CC BY 4.0 协议发布）。

```
from IPython.display import Audio
Audio('data/nightingale.wav')
```



如果你阅读的是纸版书，那么就不得不展开想象力了！这段声音是这样的：嘀哩哩哩哩嘀哩哩嘀哩哩哩哩哩。

因为不是所有人都能流利地说鸟语，所以如果能用图形将这段声音信号表示出来，那么效果可能会更好。

先载入音频文件，其中包括采样率（每秒的测量次数）和音频数据，音频数据是一个形状为 `(N, 2)` 的数组，因为是立体声，所以有 2 列。

```
from scipy.io import wavfile

rate, audio = wavfile.read('data/nightingale.wav')
```

通过求左声道和右声道的均值，将其转换为单声道。

```
audio = np.mean(audio, axis=1)
```

然后计算声音的长度，并绘制出音频数据（见图 4-2）。

```
N = audio.shape[0]
L = N / rate

print(f'Audio length: {L:.2f} seconds')

f, ax = plt.subplots()
ax.plot(np.arange(N) / rate, audio)
ax.set_xlabel('Time [s]')
ax.set_ylabel('Amplitude [unknown]');

Audio length: 7.67 seconds
```

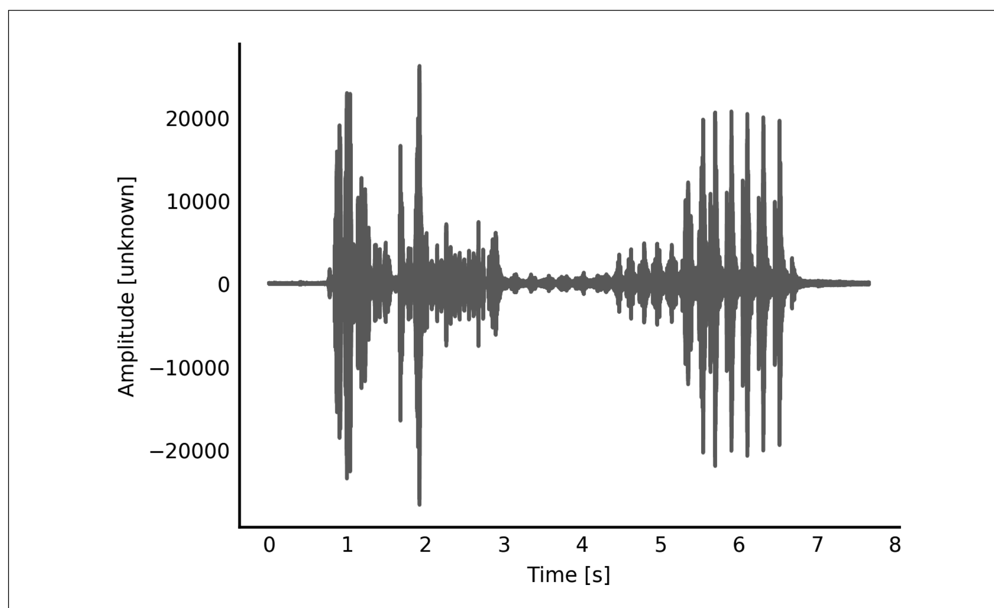


图 4-2：夜莺鸣叫的音频波形图

还是不太令人满意，对吗？如果将这样的电压输入喇叭，或许可以听到鸟儿的鸣叫，但我们无法在头脑中具体想象出这种声音。有没有更好的表示方法呢？

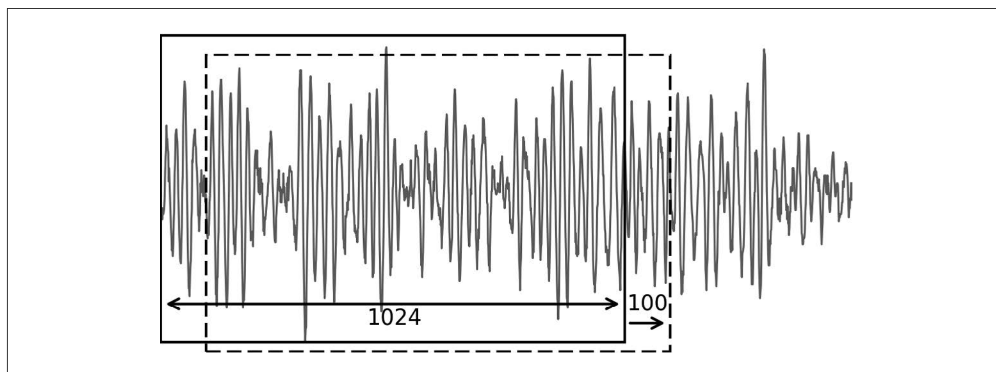
有的，那就是离散傅里叶变换，其中**离散**指的是这段录音是由有时间间隔的声音测量数据组成的。与之相对的是连续录音，比如磁带（还记得那种盒式录音带吗）。离散傅里叶变换经常用快速傅里叶变换算法来计算，在非正式场合，快速傅里叶变换可以用于指代离散傅里叶变换。离散傅里叶变换告诉哪些频率或“音符”会出现在的信号中。

当然，鸟儿的歌声中包含许多音符，因此还应该知道每个音符的发生时间。傅里叶变换在时域（即随时间进行的一组测量）中接受信号，然后将其转换为一个频谱，即一组带有相

应（复数[2]）值的频率。频谱不包含任何关于时间的信息！[3]

因此，如果想在得到频率的同时得到该频率的发生时间，需要再动点脑筋。策略如下：接受音频信号，将其分割为有重叠的小片段，然后对每个片段应用傅里叶变换（这种技术称为**短时傅里叶变换**）。

我们将信号分割为 1024 个采样片段，每个采样大概是 0.02 秒的音频。在随后检查效果时，将解释为何选择 1024 而不是 1000 个片段的原因。每个片段之间有 100 个采样的重叠，如下图所示。



将信号分割为 1024 个采样片段，每个片段与前面的片段有 100 个重叠采样。得到的 `slices` 对象每行包含一个片段。

```
from skimage import util

M = 1024

slices = util.view_as_windows(audio, window_shape=(M,), step=100)
print(f'Audio shape: {audio.shape}, Sliced audio shape: {slices.shape}')

Audio shape: (338081,), Sliced audio shape: (3371, 1024)
```

生成一个加窗函数（参见 4.6.2 节中对基本假设和每条假设的讨论），并用它乘以信号。

```
win = np.hanning(M + 1)[:-1]
slices = slices * win
```

每列一个片段更方便操作，因此进行转置。

```
slices = slices.T
print('Shape of `slices`:', slices.shape)

Shape of `slices`: (1024, 3371)
```

注 2：实际上，傅里叶变换告诉我们如何将一组频率不同的正弦波组合成输入信号。频谱由复数组成，每个复数代表一个正弦波。复数编码了两个指标：幅值和相角。幅值表示信号中的正弦波的强度，相角表示正弦波随时间变化的程度。本章只关注可以用 `np.abs` 计算的幅值。

注 3：如果想同时（近似）计算频率和发生时间，可以阅读小波分析的相关资料，以了解更多相关技术。

为每个片段计算离散傅里叶变换，返回的结果中既有正频率，又有负频率（详见 4.6.1 节），这里提取出正的 M2 频率。

```
spectrum = np.fft.fft(slices, axis=0)[:M // 2 + 1:-1]
spectrum = np.abs(spectrum)
```

（简单说明一下，你会注意到我们交替使用了 `scipy.fftpack.fft` 和 `np.fft` 函数。NumPy 提供了基础的快速傅里叶变换功能，SciPy 则对其进行了扩展，但它们之中都包含 `fft` 函数，这是基于 Fortran 语言的 FFTPACK 开发的。）

频谱中可能包含非常大的值和非常小的值，对其取对数可以显著压缩取值范围。

接下来用信号和信号最大值的比率的对数来绘图（见图 4-3），这个比率有专门的单位，称为分贝，即 $20\log_{10}$（振幅比）。

```
f, ax = plt.subplots(figsize=(4.8, 2.4))

S = np.abs(spectrum)
S = 20 * np.log10(S / np.max(S))

ax.imshow(S, origin='lower', cmap='viridis',
          extent=(0, L, 0, rate / 2 / 1000))
ax.axis('tight')
ax.set_ylabel('Frequency [kHz]')
ax.set_xlabel('Time [s]');
```

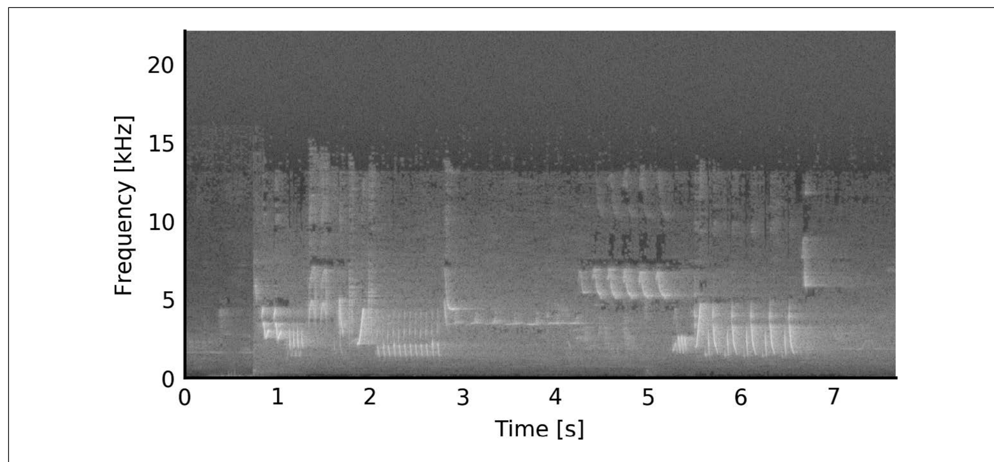


图 4-3：鸟鸣频谱图

这下好多了！现在可以看出，频率是随着时间变化的，这段频谱对应声音发出的方式。看看能否匹配前面的描述：嘀哩哩哩哩嘀哩哩嘀哩哩哩哩哩。（我就不模仿 3~5 秒的声音了，那是另一种鸟。）

SciPy 已经用 `scipy.signal.spectrogram` 实现了这个过程（见图 4-4），可以用如下方式调用。

```
from scipy import signal

freqs, times, Sx = signal.spectrogram(audio, fs=rate, window='hanning',
                                      nperseg=1024, noverlap=M - 100,
                                      detrend=False, scaling='spectrum')

f, ax = plt.subplots(figsize=(4.8, 2.4))
ax.pcolormesh(times, freqs / 1000, 10 * np.log10(Sx), cmap='viridis')
ax.set_ylabel('Frequency [kHz]')
ax.set_xlabel('Time [s]');
```

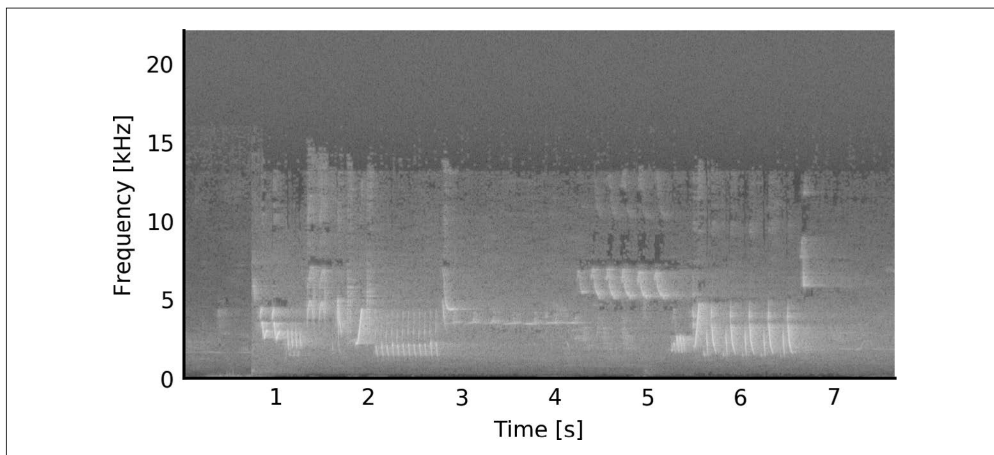


图 4-4：SciPy 内置函数生成的鸟鸣频谱图

手动建立的频谱函数与 SciPy 内置函数的唯一区别是，SciPy 返回的是频谱幅值的平方（将测量电压变成了测量能量），并乘以某个标准化因子。[4]

4.3 历史

傅里叶变换的确切起源很难寻找，一些相关方法甚至可以回溯到巴比伦时代。但在 19 世纪初期，计算行星轨道和求解热力学（流体）方程是非常热门的话题，并取得了一些重大突破。克莱罗、拉格朗日、欧拉、高斯和达朗贝尔这些人中，究竟谁对傅里叶变换做出了重大贡献，尚没有定论；但第一个描述快速傅里叶变换（一种计算离散傅里叶变换的算法，Cooley 和 Tukey 于 1965 年普及了这种方法）的人是高斯。约瑟夫·傅里叶（这种变换就是以其名字命名的）首先提出，**任意周期**[5] 的函数都可以表示为多个三角函数的和。

注 4：SciPy 为在频谱中保留能量做了一些努力。因此，当只取一半分量（如 N 个偶分量）时，它会将除第一个和最后一个分量（这两个分量由频谱的两半“共享”）以外的其余分量乘以 2。它还会对窗口进行标准化，方式是用窗口除以窗口总数。

注 5：实际上，周期可以是无穷大的！广义连续傅里叶变换提供了这种可能。离散傅里叶变换通常定义在一个有限区间上，这个区间就隐式地定义了要进行变换的时域函数的周期。换句话说，如果执行了反向离散傅里叶变换，就一定能得到一个周期性信号。

4.4 实现

SciPy 在 `scipy.fftpack` 模块中提供了离散傅里叶变换功能。除此之外，它还提供了以下与离散傅里叶变换相关的功能。

- `fft`、`fft2`、`fftn`
  在一维、二维和 n 维空间中用快速傅里叶变换算法计算离散傅里叶变换。
- `ifft`、`ifft2`、`ifftn`
  计算反向离散傅里叶变换。
- `dct`、`idct`、`dst`、`idst`
  计算余弦和正弦变换，及其反变换。
- `fftshift`、`ifftshift`
  `fftshift` 将零频分量平移到频谱中心，`ifftshift` 则撤销 `fftshift` 的效果（稍后会做更多介绍）。
- `fftfreg`
  返回离散傅里叶变换采样频率。
- `rfft`
  计算一个实数序列的离散傅里叶变换，它用结果频谱的对称性来提高性能，应用时实际上使用的还是 `fft`。

NumPy 中的以下函数是对这个列表的良好补充：`np.hanning`、`np.hamming`、`np.bartlett`、`np.blackman`、`np.kaiser`。它们都是加窗函数。

还可以通过 `scipy.signal.fftconvolve` 用离散傅里叶变换对大量输入执行快速卷积操作。

SciPy 包装了 Fortran 的 FFTPACK 库——它不是最快的，但和 FFTW 这些包不同，它有宽松式的免费软件许可。

4.5 选择离散傅里叶变换的长度

基本的离散傅里叶变换计算需要 $O(N^2)$ 个操作。[6] 为什么呢？你有 N（复数）个具有不同频率的正弦波（$2\pi f\times 0$，$2\pi f\times 1$，$2\pi f\times 2$，…，$2\pi f\times(N-1)$），而且你还想知道信号间的彼此联系有多强。从第一个信号开始，你要用信号做点积（这一步就需要 N 次乘法操作）。这个操作要重复 N 次，每个正弦波都要做一次，因此需要 N^2 次操作。

对比之下，快速傅里叶变换在理想情况下是 $O(N\log N)$，因为它可以聪明地重用计算，这是一项重大改进！但是，FFTPACK 中实现的经典 Cooley-Tukey 算法（SciPy 使用的就是

注 6：在计算机科学中，算法的计算成本通常用“大 O”表示法来表示。这种表示法告诉我们算法运行时间是如何随着元素数目的增长而增加的。如果一个算法是 $O(N)$ 的，那么其运行时间随着输入元素数目而线性增加（例如，在未排序列表中搜索特定值是 $O(N)$ 的）。冒泡排序是 $O(N^2)$ 算法的一个示例，实际执行的运算数目理论上是 $N+1/2N^2$，这表明算法的计算成本是随着输入元素数目的平方而增加的。

这种算法）递归地将变换过程分解成很多更小的（质数长度）变换过程，这种改进只能在“平滑”的输入长度上表现出来（当输入长度的最大质因数很小时，就认为其是平滑的，如图 4-5 所示）。对于较大质数长度的过程，可以将 Bluestein 或 Radar 算法与 Cooley-Tucky 算法配合使用，但 FFTPACK 中没有实现这种优化。[7]

接下来演示一下。

```
import time

from scipy import fftpack
from sympy import factorint

K = 1000
lengths = range(250, 260)

# 计算所有输入长度的平滑度
smoothness = [max(factorint(i).keys()) for i in lengths]


exec_times = []
for i in lengths:
    z = np.random.random(i)
    # 对于每个输入长度i，执行K次快速傅里叶变换，并保存运行时间

    times = []
    for k in range(K):
        tic = time.monotonic()
        fftpack.fft(z)
        toc = time.monotonic()
        times.append(toc - tic)

    # 对于每个输入长度，记录下最短的执行时间
    exec_times.append(min(times))


f, (ax0, ax1) = plt.subplots(2, 1, sharex=True)
ax0.stem(lengths, np.array(exec_times) * 10**6)
ax0.set_ylabel('Execution time (µs)')

ax1.stem(lengths, smoothness)
ax1.set_ylabel('Smoothness of input length\n(lower is better)')
ax1.set_xlabel('Length of input');
```

注 7：虽然最好不要重新实现现有算法，但有时为了获得尽可能快的执行速度，还是需要重新实现。比如，Cython 可以将 Python 编译为 C，Numba 可以实时编译 Python 代码，这些工具可以使我们的工作更易完成（速度也更快）。如果能够使用 GPL 许可证软件，那么可以考虑用 PyFFTW 来执行快速傅里叶变换。

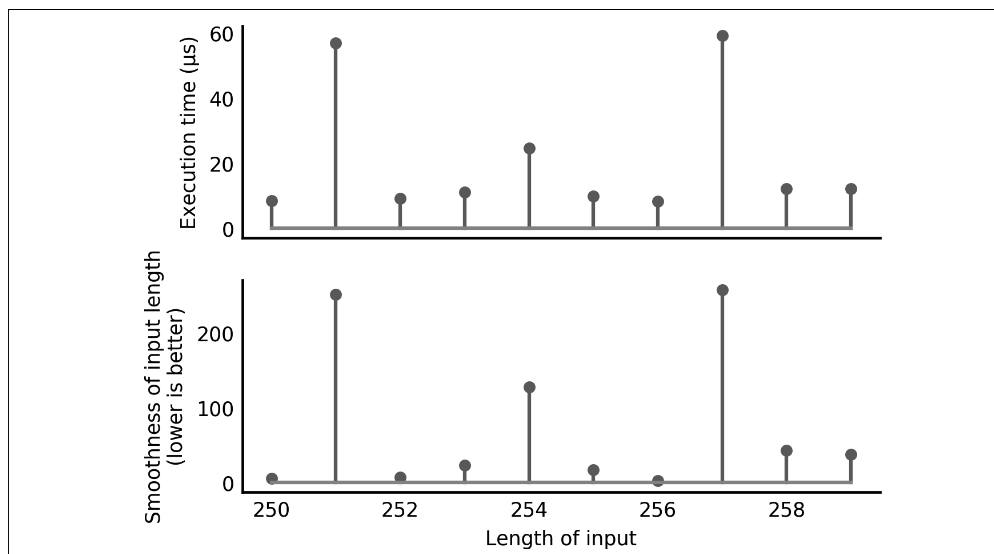


图 4-5：快速傅里叶变换执行时间与不同输入长度的平滑度

Cooley-Tukey 算法的原理是，可以将长度为平滑数字的快速傅里叶变换分解为多个较小的子变换，完成第一个子变换后，就可以在随后的计算中重用其结果。这就解释了为什么要在前面的音频分段中选择 1024 的长度，它的平滑度只有 2，因此可以使用最理想的“基 2 Cooley-Tukey”算法。在计算快速傅里叶变换时，这种方法只使用 $(N/2)\log_2 N = 5120$ 次复数乘法，而不是 $N^2 = 1\ 048\ 576$ 次。选择 $N = 2^m$ 可以确保一个最为平滑的 N（因此可以进行最快的快速傅里叶变换）。

4.6 更多离散傅里叶变换概念

在进行比较复杂的傅里叶变换前，先介绍几个需要了解的常用概念，然后解决一个实际问题：分析雷达数据中的目标检测。

4.6.1 频率及其排序

由于历史原因，多数算法实现都返回一个数组，其中的频率高低起伏（参见本节中的附注栏“离散傅里叶变换”，获取有关频率的更多解释）。例如，当进行一个信号值都为 1 的实数傅里叶变换时，输入是一直不变的，因此只在数组的第一项中有一个几乎不变的傅里叶分量（也称作 direct current，“DC”分量或直流电，即电学中的术语“信号平均值”）。

```
from scipy import fftpack
N = 10

fftpack.fft(np.ones(N)) # 第一个分量是np.mean(x) * N

array([ 10.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,   0.+0.j,
         0.-0.j,   0.-0.j,   0.-0.j,   0.-0.j])
```

当在快速变化的信号中使用快速傅里叶变换时，可以看到出现了一个高频分量。

```
z = np.ones(10)
z[::2] = -1

print(f'Applying FFT to {z}')
fftpack.fft(z)

Applying FFT to [-1.  1.  -1.  1.  -1.  1.  -1.  1 . -1.  1.]

array([  0.+0.j,   0.+0.j,    0.+0.j,    0.+0.j,     0.+0.j,   -10.+0.j,
         0.-0.j,   0.-0.j,    0.-0.j,    0.-0.j])
```

注意，在输入实数的情况下，快速傅里叶变换返回一个共轭对称的复数频谱（即实数部分对称，虚数部分反对称）。

```
x = np.array([1, 5, 12, 7, 3, 0, 4, 3, 2, 8])
X = fftpack.fft(x)

np.set_printoptions(precision=2)

print("Real part:     ", X.real)
print("Imaginary part:", X.imag)

np.set_printoptions()

Real part:      [ 45.     7.09 -12.24   -4.09   -7.76   -1.   -7.76  -4.09 -12.24
                  7.09]
Imaginary part: [  0.  -10.96   -1.62   12.03    6.88    0.   -6.88 -12.03   1.62
                 10.96]
```

（再次强调，第一个分量是 `np.mean(x) * N`。）

`fftfreq` 函数显示哪个频率需要特别关注。

```
fftpack.fftfreq(10)

array([ 0. ,  0.1,  0.2,  0.3,  0.4,  -0.5,  -0.4,  -0.3,  -0.2,  -0.1])
```

结果显示，最大分量发生在 0.5 个采样周期的频率上。这与输入吻合，输入是每秒采样一次的负一正一循环。

有时为了便于查看，需要稍微重组一下频谱，比如从高负位置平移到从低到高的正位置（当前不对负频率的概念做更多探讨，了解实际的正弦波由正频率和负频率的组合产生就可以了）。可以用 `fftshift` 函数来重组频谱。

离散傅里叶变换

如果可以通过时间函数 $x(t)$（或其他变量的函数，依具体应用而定）得到一个等间隔的 N 个实数或复数的采样序列 x_0，x_1，…，x_{N-1}，则离散傅里叶变换可以用以下求和公式将这个序列转换为 N 个复数 X_k 的序列。

$$X_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-\mathrm{j}2\pi kn/N},\ k = 0,\ 1,\ \cdots > N-1$$

如果已知数值 X_k，那么反向离散傅里叶变换可以用以下求和公式**精确地**还原采样值 x_n。

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{j2\pi kn/N}$$

记住 $e^{j\theta} = \cos\theta + j\sin\theta$，由后一个公式可知，离散傅里叶变换可以将序列 x_n 分解为一个系数为 X_k 的复数离散傅里叶序列。比较一下离散傅里叶变换和连续复数傅里叶序列。

$$x(t) = \sum_{n=-\infty}^{\infty} c_n e^{jn\omega_0 t}$$

离散傅里叶变换是**有限**序列，其 N 个项定义在区间 $[0, 2\pi)$（**包括 0，不包括 2π**）中**角度** $(\omega_0 t_n) = 2\pi k/N$ 的等间隔离散实例上。这样可以自动对离散傅里叶变换进行标准化，使时间不会显式地出现在正向变换或反向变换中。

如果初始函数 $x(t)$ 的频率被限制为小于采样频率的一半（即所谓的**奈奎斯特频率**，Nyquist frequency），那么由反向离散傅里叶变换产生的采样值之间的插值通常可以准确地重建 $x(t)$。如果 $x(t)$ **不满足**这个限制，那么通常不能用反向离散傅里叶变换通过插值重建 $x(t)$。注意，这个限制并不意味着**没有**方法进行这种重建，因为还可以使用压缩感知或有限创新率抽样等方法。

函数 $e^{j2\pi k|N} = (e^{j2\pi|N})^k = w^k$ 取的是复平面中单位圆上 0 和 $2\pi(N-1)/N$ 之间的离散值。函数 $e^{j2\pi kn|N} = w^{kn}$ 围绕原点旋转 $n(N-1)/N$ 次，产生出 $n = 1$ 的基础正弦波的谐波。

对于偶数 N，当 $n>N/2$ 时，定义离散傅里叶变换的方式导致了一些微妙之处。[8] 图 4-6 中的函数 $e^{j2\pi kn|N}$ 是按 k 值增加的方式绘制的，表示的情形是从 $n = 1$ 到 $n = N-1$，$N = 16$。

当 k 增加到 $k + 1$ 时，角度增加了 $2\pi n/N$。当 $n = 1$ 时，步长为 $2\pi/N$。当 $n = N - 1$ 时，角度每次增加 $2\pi(N - 1)/N = 2\pi - 2\pi/N$。因为 2π 正好是绕圆一周，所以步长等于 $-2\pi/N$，实际上是个负频率方向。小于 $N/2$ 的分量表示**正**频率分量，而那些大于 $N/2$ 小于 $N - 1$ 的分量则表示**负**频率分量。对于偶数 N，$N/2$ 分量的角度增量对于每个 k 的增量来说都是正好绕一个半圆，因此既可以认为它是一个正频率，也可以认为它是一个负频率。这个离散傅里叶变换的分量表示奈奎斯特频率（即采样频率的一半），可用于在查看离散傅里叶变换图时做自我定位。

快速傅里叶变换只是一种计算离散傅里叶变换的特殊而高效的算法。直接计算离散傅里叶变换所需的计算量是 N^2 级别，而快速傅里叶变换算法的计算量只有 $N \log N$。快速傅里叶变换对离散傅里叶变换在实时应用中的广泛普及起到了关键作用，2000 年被 *IEEE Journal Computing in Science & Engineering* 评选为 20 世纪十大算法之一。

注 8：我们给你留的练习是，想象一下 N 为奇数的情况。本章中的所有示例使用的都是偶数次离散傅里叶变换。

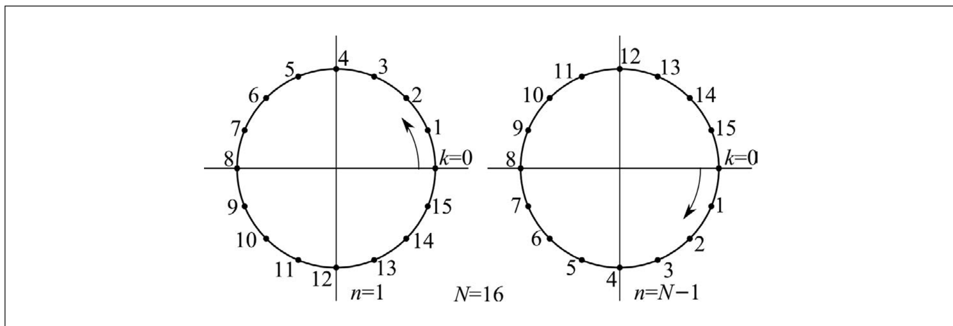


图 4-6：单位圆采样

检查一下带噪声的图像（见图 4-7）中的频率分量。注意，虽然一幅静态图像中没有随时间变化的分量，但其值却可以随**空间**变化。离散傅里叶变换既可以应用于时间，也可以应用于空间。

首先，载入并显示图像。

```
from skimage import io
image = io.imread('images/moonlanding.png')
M, N = image.shape

f, ax = plt.subplots(figsize=(4.8, 4.8))
ax.imshow(image)

print((M, N), image.dtype)

(474, 630) uint8
```

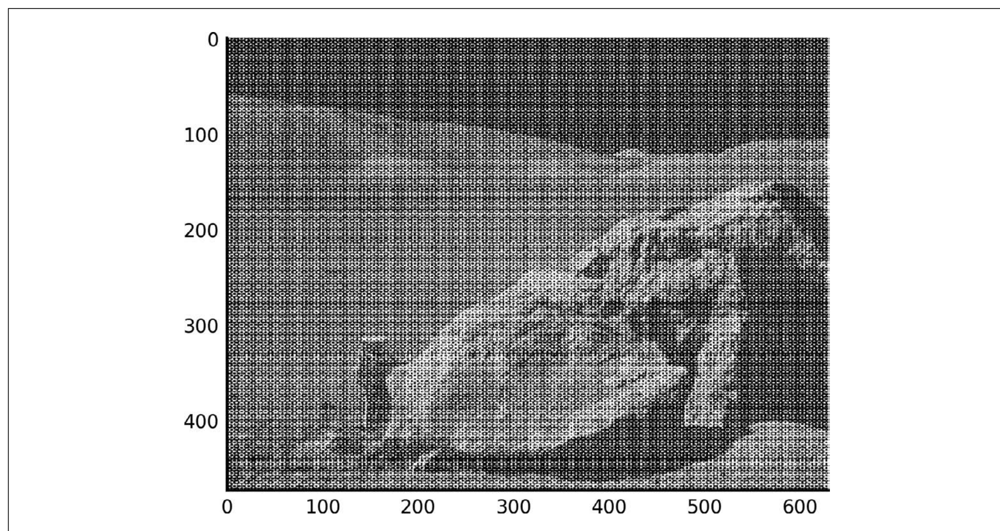


图 4-7：一幅带噪声的登月图像

不要调整你的显示器！你看到的就是真实的图像，只不过要么是拍摄设备的问题，要么是传输设备的问题，图像显然被扭曲变形了。

为了检查这幅图像中的频谱，要使用 fftn（不是 fft）来计算离散傅里叶变换，因为它有不止一个维度。二维快速傅里叶变换等价于先在行上进行一维快速傅里叶变换，然后再在列上进行一维快速傅里叶变换，反之亦可。

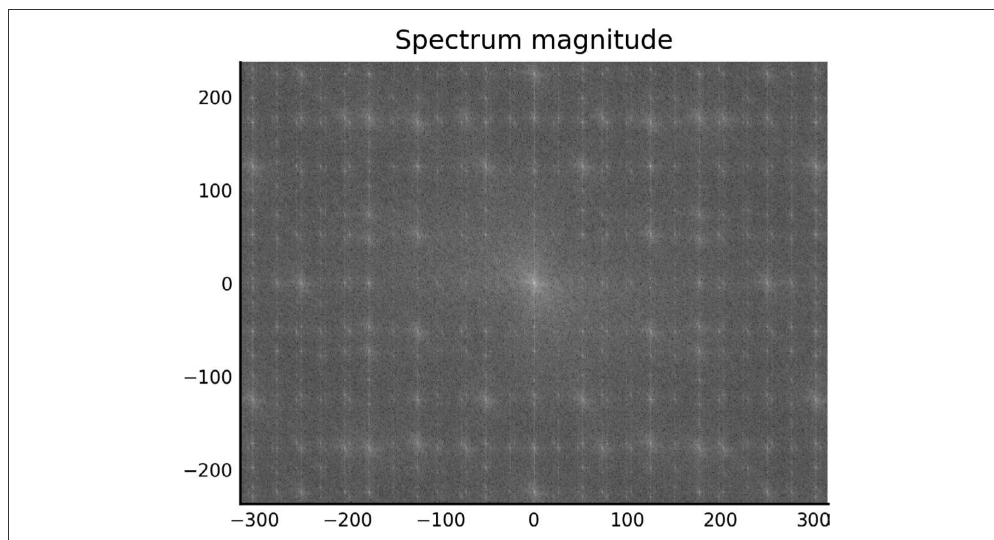
```
F = fftpack.fftn(image)

F_magnitude = np.abs(F)
F_magnitude = fftpack.fftshift(F_magnitude)
```

同样，在显示频谱前，要取其对数来压缩值域。

```
f, ax = plt.subplots(figsize=(4.8, 4.8))

ax.imshow(np.log(1 + F_magnitude), cmap='viridis',
          extent=(-N // 2, N // 2, -M // 2, M // 2))
ax.set_title('Spectrum magnitude');
```



注意频谱原点（中心）周围的高值，这些系数描述了图像的低频或平滑部分，是照片中模糊的画布。高频分量分布在整个频谱中，充满了边边角角。高频峰值对应周期性的噪声。

从这张照片中可以看出，（测量仪器造成的）噪声是高度周期化的，因此我们希望通过删除频谱中的相应部分以去除噪声（见图 4-8）。

抑制了峰值后，图像看起来确实不一样了！

```
# 将频谱中心的一个块区归零
K = 40
F_magnitude[M // 2 - K: M // 2 + K, N // 2 - K: N // 2 + K] = 0

# 找出高于第98个百分位数的所有峰值
```

```
peaks = F_magnitude < np.percentile(F_magnitude, 98)

# 将峰值平移回去，靠近原来的频谱
peaks = fftpack.ifftshift(peaks)

# 制作一个原始（复数）频谱的副本
F_dim = F.copy()

# 将这些峰值的系数设定为0
F_dim = F_dim * peaks.astype(int)

# 执行反向傅里叶变换以还原图像
# 因为从一个实数图像开始，所以只检查输出的实数部分
image_filtered = np.real(fftpack.ifft2(F_dim))

f, (ax0, ax1) = plt.subplots(2, 1, figsize=(4.8, 7))
ax0.imshow(np.log10(1 + np.abs(F_dim)), cmap='viridis')
ax0.set_title('Spectrum after suppression')

ax1.imshow(image_filtered)
ax1.set_title('Reconstructed image');
```

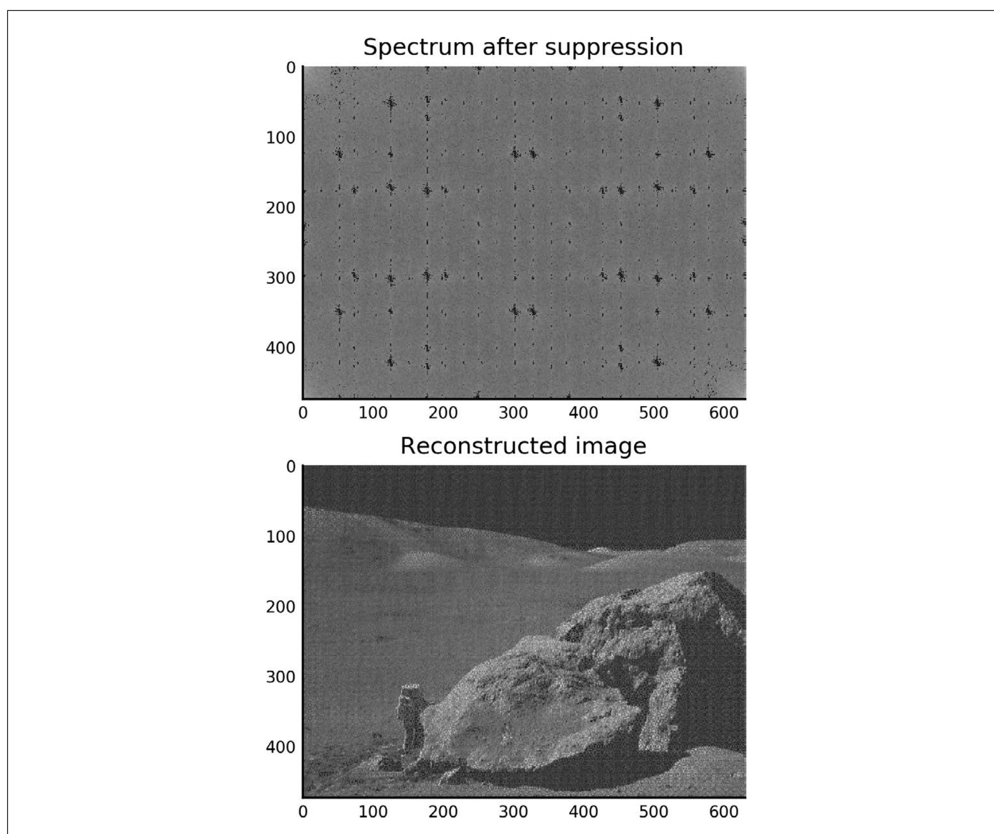


图 4-8：滤波后的登月图像及其频谱

4.6.2 加窗

如果检查矩形脉冲的傅里叶变换，可以看到频谱中有许多旁瓣。

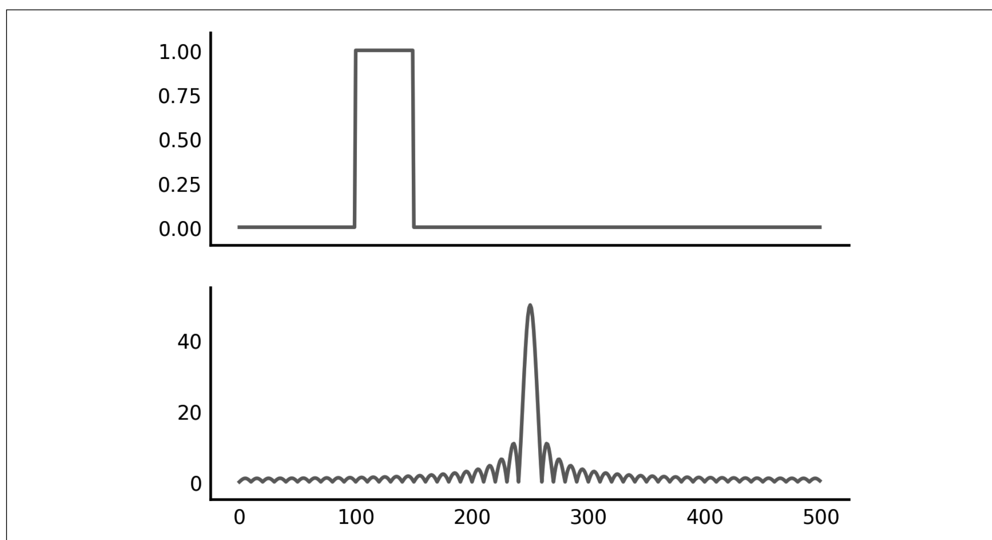
```
x = np.zeros(500)
x[100:150] = 1

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1, sharex=True)

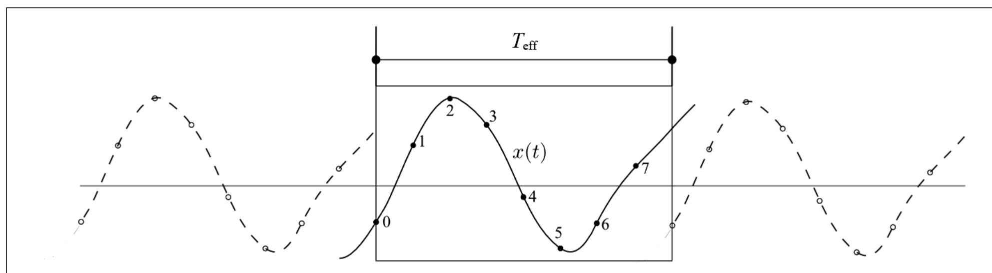
ax0.plot(x)
ax0.set_ylim(-0.1, 1.1)

ax1.plot(fftpack.fftshift(np.abs(X)))
ax1.set_ylim(-5, 55);
```



从理论上来说，要想表示这种突变信号，你需要一个无限多的正弦波（频率）组合。它们的系数也有典型的旁瓣结构，就像上面的脉冲一样。

重要的是，离散傅里叶变换假设输入信号是周期性的。如果信号不是周期性的，那么这个假设可以简单地认为，在信号的末端，它会跳回到开始的值。思考一下函数 $x(t)$，如下所示。



我们只在短时间内测量信号，记为 T_{eff}。傅里叶变换假设 $x(8) = x(0)$，信号像虚线那样继续，而不是沿着实线。这样就在边缘产生了一个大跳跃，并在频谱中不出所料地引起了振荡。

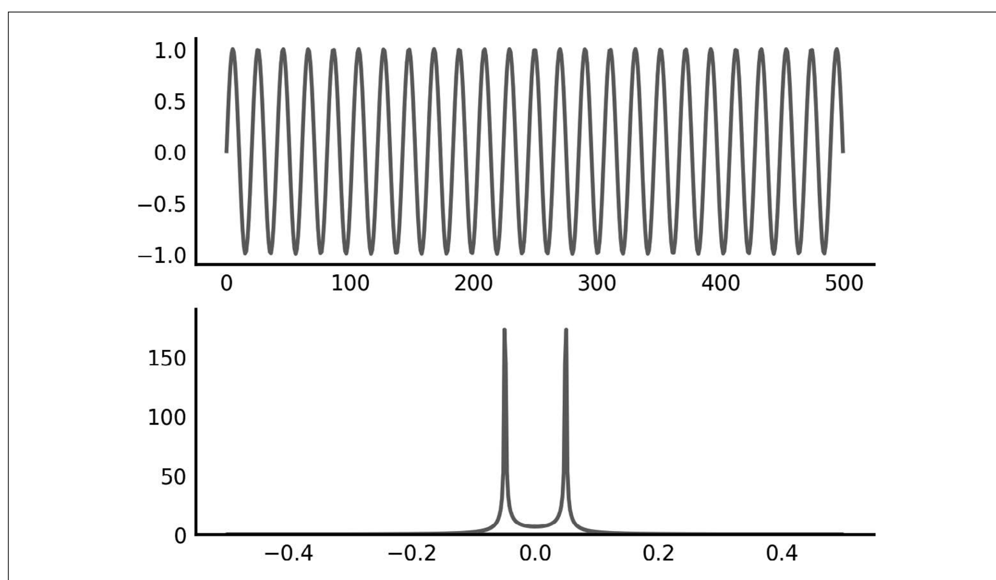
```
t = np.linspace(0, 1, 500)
x = np.sin(49 * np.pi * t)

X = fftpack.fft(x)

f, (ax0, ax1) = plt.subplots(2, 1)

ax0.plot(x)
ax0.set_ylim(-1.1, 1.1)

ax1.plot(fftpack.fftfreq(len(t)), np.abs(X))
ax1.set_ylim(0, 190);
```



不像预想的那样有两条线，频谱中充满了波峰。

可以用**加窗**（windowing）处理来消除这种效果，即将初始函数乘以一个窗口函数，如 Kaiser 窗口 $K(N, \beta)$。下面画出 β 范围为 0~100 的 Kaiser 窗口函数。

```
f, ax = plt.subplots()

N = 10
beta_max = 5
colormap = plt.cm.plasma

norm = plt.Normalize(vmin=0, vmax=beta_max)

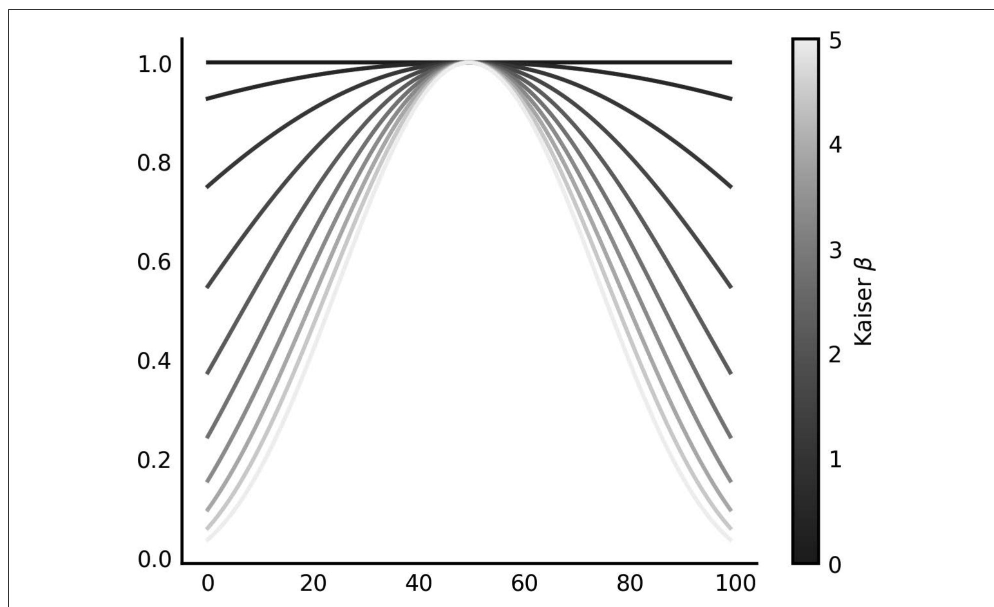
lines = [
    ax.plot(np.kaiser(100, beta), color=colormap(norm(beta)))
    for beta in np.linspace(0, beta_max, N)
```

```
    ]

sm = plt.cm.ScalarMappable(cmap=colormap, norm=norm)

sm._A = []

plt.colorbar(sm).set_label(r'Kaiser $\beta$');
```



通过改变参数 β，可以改变窗口的形状，从矩形（$\beta = 0$，无加窗效果）窗口到能生成信号的窗口，这些信号可以在采样区间的两个端点平滑地从 0 开始增加，再逐渐减少为 0，这样的窗口能产生非常低的旁瓣（典型的 β 值在 5~10 范围内）。[9]

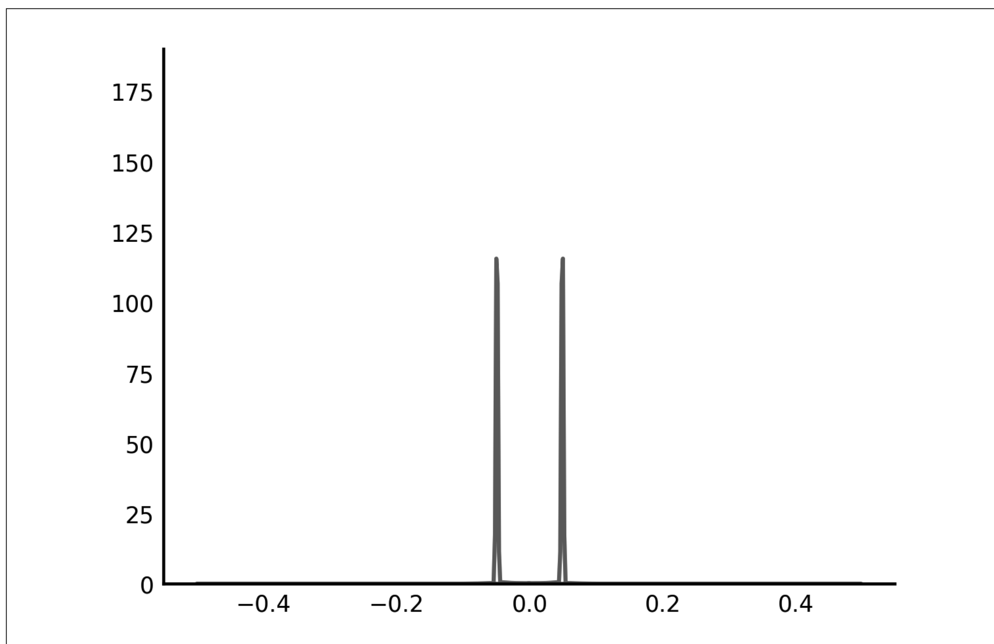
通过应用 Kaiser 窗口，可以看到旁瓣消失殆尽，但代价是主瓣有一点变宽。

对于以上示例，加窗效果立竿见影。

```
win = np.kaiser(len(t), 5)
X_win = fftpack.fft(x * win)

plt.plot(fftpack.fftfreq(len(t)), np.abs(X_win))
plt.ylim(0, 190);
```

注 9：经典的加窗函数包括 Hann、Hamming 和 Blackman。它们的不同之处在于旁瓣的水平和主瓣的宽度（在傅里叶域中）。Kaiser 窗口是一种现代的灵活的窗口函数，对于多数应用来说，它都是近似最优的。它是对最优椭球窗口的良好近似，将大多数能量都集中在主瓣上。像原文中演示的那样，通过调整参数 β，可以调整 Kaiser 窗口，使其适合具体的应用。



4.7 实际应用：分析雷达数据

线性调制的调频连续波（FMCW，frequency-modulated continuous-wave）雷达大量使用了快速傅里叶变换算法进行信号处理，并提供了各种各样的快速傅里叶变换应用示例。我们将用来自于 FMCW 雷达的真实数据演示一个快速傅里叶变换应用：目标检测。

FMCW 雷达的工作原理大致如下所示（更多信息参见本节的附注栏“简单的 FMCW 雷达系统”和图 4-9）。

(1) 变频信号产生后会被雷达天线发射出去，然后沿着远离雷达的方向向外传输。当碰到某个物体时，部分信号被反射回雷达。接收到反射信号后，雷达将信号乘以一个发射信号的副本，并进行采样，将其转换为数值，并保存在一个数组中。我们的任务就是解释这些数值，以得到有意义的结果。

(2) 上面的相乘步骤非常重要。回忆一下在学校里学过的三角恒等式。

$$\sin(xt)\sin(yt) = \frac{1}{2}\left[\sin\left((x-y)t+\frac{\pi}{2}\right)-\sin\left((x+y)t+\frac{\pi}{2}\right)\right]$$

(3) 因此，如果用发射信号乘以接收信号，就可以预测在频谱中出现两个频率分量：一个是接收信号和发射信号的频率之差，另一个是它们的频率之和。

(4) 我们尤其对两个信号的频率之差感兴趣，因为它告诉我们信号反射回雷达需要多长时间（换句话说，物体离我们有多远）的一些信息。通过对信号应用低通滤波器（即可以滤掉高频的滤波器），可以丢弃其他信息。

简单的 FMCW 雷达系统

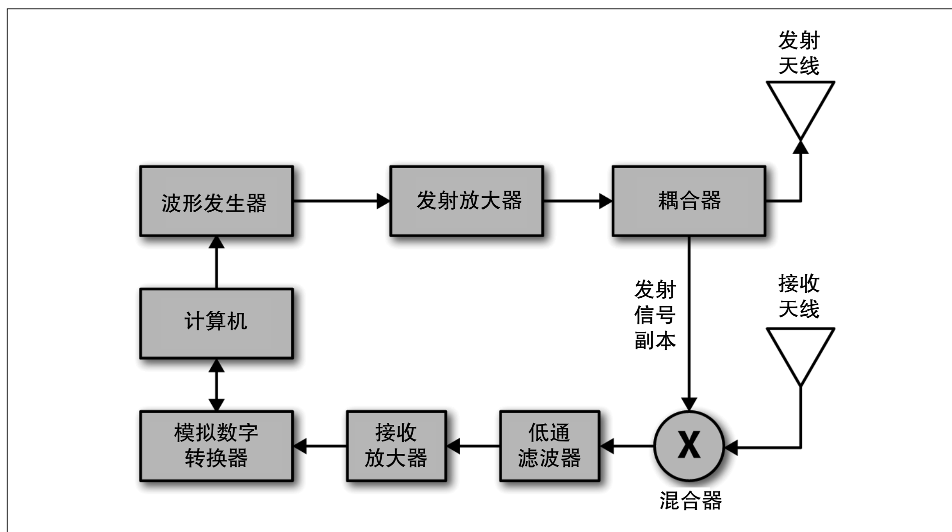


图 4-9：一个简单的 FMCW 雷达系统结构图

上面给出了一个使用独立发射天线和接收天线的简单 FMCW 雷达结构图。雷达包括一个可以产生正弦信号的波形发生器，信号的频率围绕所需频率线性变化。产生的信号被发射放大器放大到所需要的功率等级，并经由耦合器电路传递给发射天线，耦合器中还能输出一份发射信号的副本。发射天线以窄射速电磁波的形式向需要探测的目标发送发射信号。当电磁波遇到能反射电磁波的物体时，照射到目标上的一部分能量被反射回接收系统，成为向雷达系统方向传播的另一个电磁波。当这个电磁波遇到接收天线时，接收天线收集撞击到天线上的电磁波能量，并将其转换为波动的电压，提供给混合器。混合器将接收信号与发射信息的副本相乘，产生一个频率等于发射信号和接收信号频率之差的正弦波。低通滤波器确保了接收信号是频带限制的（即不包含那些我们不关心的频率），接收放大器将信号增强到适合模拟数字转换器的振幅，最后由模拟数字转换器将数据反馈给计算机。

总结一下，应该注意以下几点。

- 最终输入计算机的数据包括 N 个使用采样率 f_s（从相乘后、滤波后的信号中）进行采样的样本。
- 返回信号的**振幅**随着**反射强度**（即由目标物体与目标和雷达间的距离决定的一个属性）的不同而变化。
- **测定频率**是对目标物体与雷达间**距离**的一种表示。

为了分析雷达数据，要生成一些人造信号，然后再研究实际雷达的输出。

雷达以 S 赫兹 / 秒的速度发射，然后不断提高其频率。经过一段时间 t 后，频率会提高到 t_S（见图 4-10）。在同一时间段内，雷达信号的传输距离为 $d=tv$ 米，其中 v 是发射信号在空气

中的速度（大致与光速相同，3×10^8 米 / 秒）。

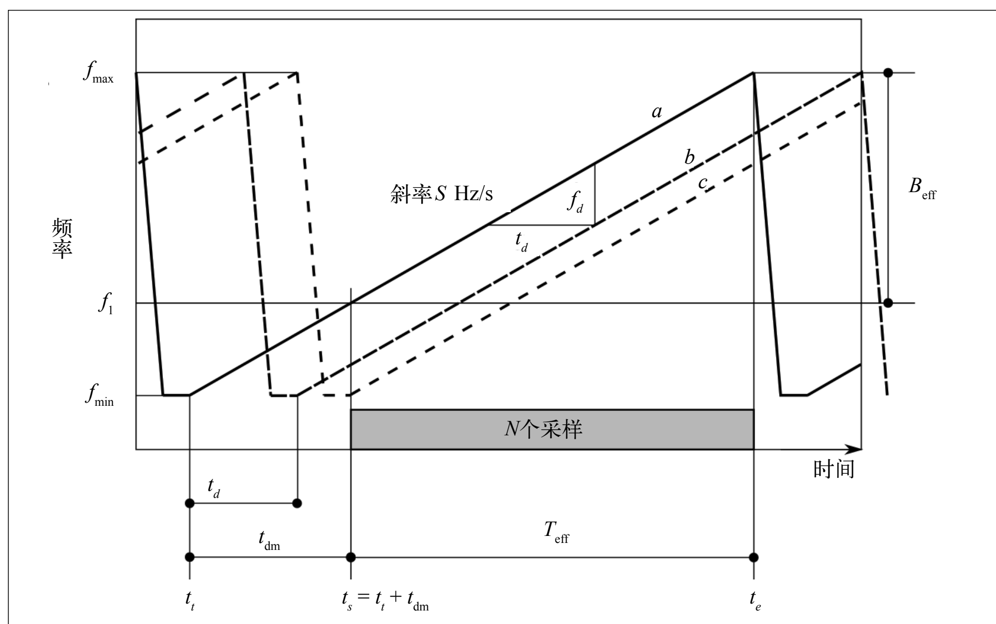


图 4-10：一个线性调频 FMCW 雷达中的频率关系

根据上图，对于一个距离为 R 的目标，可以计算出信号到达、反射和返回需要的时间。

$$t_R = 2R/v$$

```
pi = np.pi

# 雷达参数
fs = 78125          # 单位为赫兹的采样率，即每秒采样78 125次

ts = 1 / fs         # 采样时间，即每ts秒采样一次

Teff = 2048.0 * ts # 2048次采样需要的总时间（亦称有效扫描时间），单位为秒

Beff = 100e6        # 雷达采样期间的发射信号频率范围，称为“有效带宽”，单位为赫兹

S = Beff / Teff     # 频率扫描率，单位为赫兹/秒

# 目标的具体说明。我们虚构了一些目标，假设它们是能被雷达探测到的物体，具有特定的距
# 离和大小

R = np.array([100, 137, 154, 159,  180]) # 距离（单位为米）
M = np.array([0.33, 0.2, 0.9, 0.02, 0.1]) # 目标大小
P = np.array([0, pi / 2, pi / 3, pi / 5, pi / 6]) # 随机选择的相位差

t = np.arange(2048) * ts  # 采样时间

fd = 2 * S * R / 3E8      # 这些目标的频率差
```

```
# 生成5个目标
signals = np.cos(2 * pi * fd * t[:, np.newaxis] + P)

# 保存与第1个目标相关的信号，用于后续观察
v_single = signals[:, 0]

# 按照目标大小给信号加权相加，以生成能被雷达探测到的组合信号
v_sim = np.sum(M * signals, axis=1)

## 以上代码等价于
#
# v0 = np.cos(2 * pi * fd[0] * t)
# v1 = np.cos(2 * pi * fd[1] * t + pi / 2)
# v2 = np.cos(2 * pi * fd[2] * t + pi / 3)
# v3 = np.cos(2 * pi * fd[3] * t + pi / 5)
# v4 = np.cos(2 * pi * fd[4] * t + pi / 6)
#
## 综合起来
# v_single = v0
# v_sim = (0.33 * v0) + (0.2 * v1) + (0.9 * v2) + (0.02 * v3) + (0.1 * v4)
```

这样就生成了一个人造信号 V_{single}，探测单个目标时可以接收这个信号（见图 4-11）。通过计算给定时间段内的循环数量，可以计算出信号的频率，并由此算出目标距离。

然而，真实雷达几乎不可能只接收一个反射信号。模拟信号 V_{sim} 展示了有 5 个不同距离的目标（其中包括两个彼此很接近的目标，分别位于 154 米和 159 米）的雷达信号的形式，$V_{actual}(t)$ 展示了一个真实雷达的输出信号。将多个回波加在一起时，结果几乎不具有直观的意义（见图 4-11），除非通过离散傅里叶变换的视角对其更加仔细地审视。

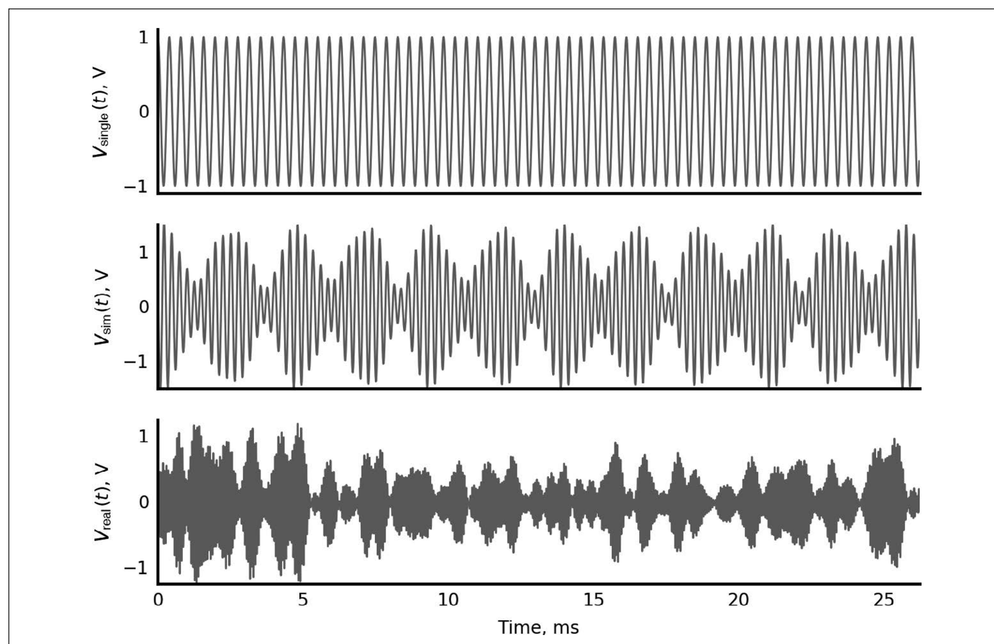


图 4-11：接收器输出信号，从上到下：单个模拟目标；5 个模拟目标；实际雷达数据

真实的雷达数据从一个 NumPy 格式的 .npz 文件（一种轻量级、跨平台、跨版本兼容的存储格式）中读取。可以用 `np.savez` 和 `np.savez_compressed` 函数保存这种文件。注意，SciPy 的 `io` 子模块还可以非常轻松地读取其他格式的文件，如 MATLAB 和 NetCDF 文件。

```
data = np.load('data/radar_scan_0.npz')

# 从radar_scan_0.npz文件中加载变量scan
scan = data['scan']

# 数据集中包含多个测量结果，每个结果由指向不同方向的雷达给出。这里取一个在特定方位角
# （左 - 右位置）和仰角（上 - 下位置）上的测量结果，其形状为(2048,)

v_actual = scan['samples'][5, 14, :]

# 信号的振幅范围是 -2.5V~+2.5V。雷达中的14位模拟数字转换器可以给
# 出 -8192~8192的整数值。通过乘以(2.5 / 8192)，可以转换回电压
v_actual = v_actual * (2.5 / 8192)
```

因为 .npz 文件可以保存多个变量，所以必须从中选择一个：`data['scan']`。它返回一个包含以下字段的**结构化 NumPy 数组**。

- `time`
  64 位（8 字节）无符号整数（`np.uint64`）
- `size`
  32 位（4 字节）无符号整数（`np.uint32`）
- `position`
  - `az`
    32 位浮点数（`np.float32`）
  - `el`
    32 位浮点数（`np.float32`）
  - `region_type`
    8 位（1 字节）无符号整数（`np.uint8`）
  - `region_id`
    16 位（2 字节）无符号整数（`np.uint16`）
  - `gain`
    8 位（1 字节）无符号整数（`np.uint16`）
  - `samples`
    2048 个 16 位（2 字节）无符号整数（`np.uint16`）

虽然 NumPy 数组确实是**同质的**（即其中所有元素都是同一类型），但这并不意味着这些元素不能是复合元素，就像这个示例一样。

可以用字典语法来访问独立的字段。

```
azimuths = scan['position']['az'] # 得到所有方位角测量结果
```

总结一下目前的情况：测量结果（V_{sim} 和 V_{actual}）是若干个目标反射回的正弦信号的总和，需要确定这些复合雷达信号的信号成分。快速傅里叶变换就是我们使用的工具。

4.7.1 频域中的信号性质

首先，对 3 个信号（单个人造目标、多个人造目标、实际目标）进行快速傅里叶变换，然后显示出正频率分量（即分量 0~N/2，见图 4-12）。用雷达术语描述，就是**距离跟踪**。

```
fig, axes = plt.subplots(3, 1, sharex=True, figsize=(4.8, 2.4))

# 对信号进行快速傅里叶变换；注意惯例，在命名快速傅里叶变换对象时，首字母应大写

V_single = np.fft.fft(v_single)
V_sim = np.fft.fft(v_sim)
V_actual = np.fft.fft(v_actual)

N = len(V_single)

with plt.style.context('style/thinner.mplstyle'):
    axes[0].plot(np.abs(V_single[:N // 2]))
    axes[0].set_ylabel("$|V_\mathrm{single}|$")
    axes[0].set_xlim(0, N // 2)
    axes[0].set_ylim(0, 1100)

    axes[1].plot(np.abs(V_sim[:N // 2]))
    axes[1].set_ylabel("$|V_\mathrm{sim} |$")
    axes[1].set_ylim(0, 1000)

    axes[2].plot(np.abs(V_actual[:N // 2]))
    axes[2].set_ylim(0, 750)
    axes[2].set_ylabel("$|V_\mathrm{actual}|$")

    axes[2].set_xlabel("FFT component $n$")

    for ax in axes:
        ax.grid()
```

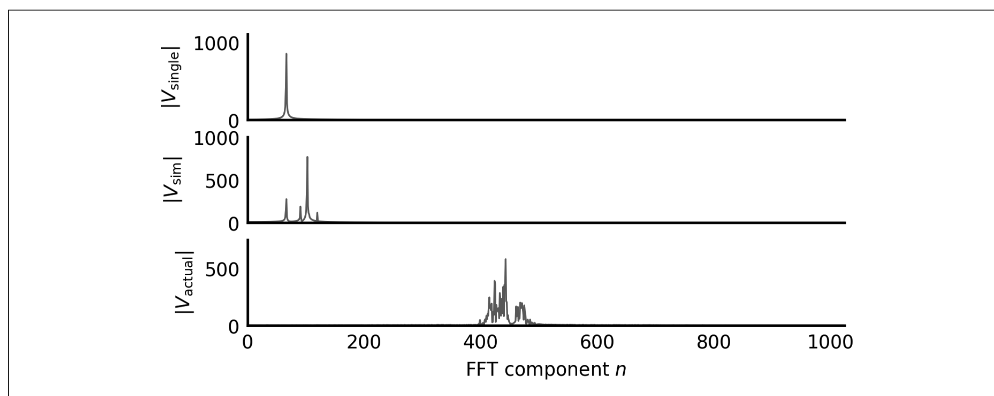


图 4-12：距离跟踪，从上到下：单个模拟目标；多个模拟目标；真实目标

信息突然变得有意义了。

根据 $|V_0|$[10] 的图形，可以清楚地看出分量 67 处有一个目标。$|V_{\text{sim}}|$ 的图形显示了多个目标，它们产生的信号无法在时域中解释。真实雷达信号 $|V_{\text{actual}}|$ 显示，分量 400~500 范围内有大量目标，而且分量 443 处有一个大的峰值。这恰好是雷达信号遇到一个露天矿的高墙后返回的回波。

为了从图中得到有用的信息，必须确定距离！我们还是使用公式。

$$R_n = \frac{nv}{2B_{\text{eff}}}$$

用雷达术语来说，每个离散傅里叶变换分量都称为**距离箱**。

这个公式还定义了雷达的**距离分辨率**：只有那些由多于 2 个的距离箱分离出来的目标才是可辨识的，如下所示。

$$\Delta R > \frac{1}{B_{\text{eff}}}$$

这是所有类型的雷达的一个基本性质。

这个结果非常令人满意，但是这个动态距离的变化范围太大了，很容易错过一些波峰，因此做一下对数变换，就像前面在频谱图中所做的那样。

```
c = 3e8 # 电磁波在空气中传播的速度大致等于光速

fig, (ax0, ax1, ax2) = plt.subplots(3, 1)


def dB(y):
    "Calculate the log ratio of y / max(y) in decibel."

    y = np.abs(y)
    y /= y.max()

    return 20 * np.log10(y)


def log_plot_normalized(x, y, ylabel, ax):
    ax.plot(x, dB(y))
    ax.set_ylabel(ylabel)
    ax.grid()

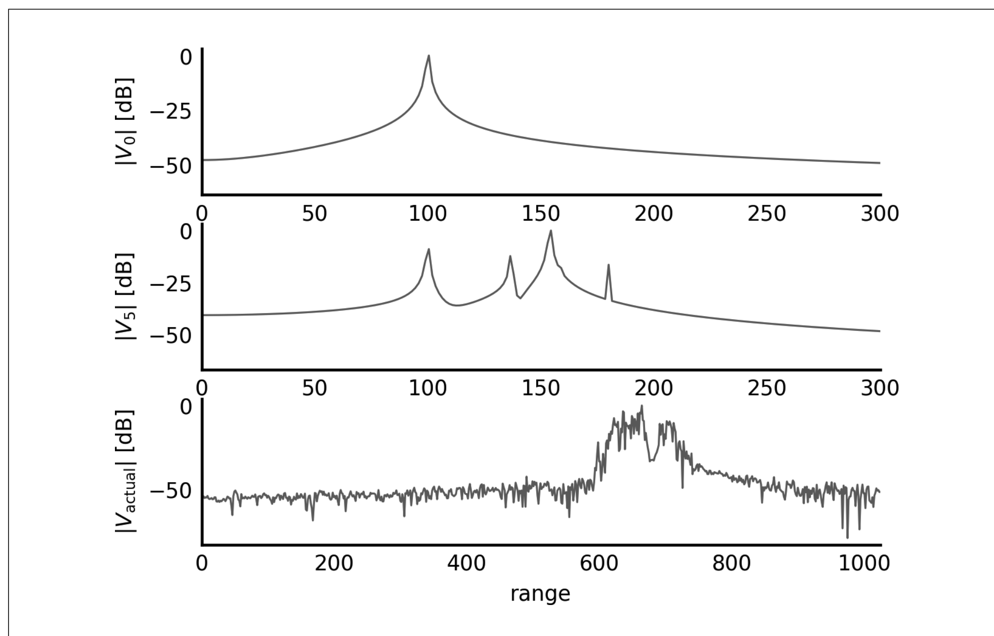

rng = np.arange(N // 2) * c / 2 / Beff

with plt.style.context('style/thinner.mplstyle'):
    log_plot_normalized(rng, V_single[:N // 2], "$|V_0|$ [dB]", ax0)
```

注 10：即图中的 $|V_{\text{single}}|$。——译者注

```
    log_plot_normalized(rng, V_sim[:N // 2], "$|V_5|$ [dB]", ax1)
    log_plot_normalized(rng, V_actual[:N // 2], "$|V_{\mathrm{actual}}|$ [dB]"
        , ax2)

ax0.set_xlim(0, 300) # 修改这些图形的x轴范围，
ax1.set_xlim(0, 300) # 以便更清楚地看到波峰形状
ax2.set_xlim(0, len(V_actual) // 2)
ax2.set_xlabel('range')
```



这些图形可以更好地说明可探测的动态距离。例如，在真实雷达信号中，雷达的**噪声基底**变得可见了（也就是说，雷达探测目标的能力受到了系统内的电子噪声水平的限制）。

4.7.2 加窗之后

虽然已经取得了很大进展，但还是没能在模拟信号的频谱中区分出 154 米和 159 米处的波峰。谁知道我们在真实信号中错过了什么呢！为了凸显出波峰，再看看其他方法，使用一下**加窗函数**。

以下示例中的信号通过 $\beta = 6.1$ 的 Kaiser 函数进行了加窗。

```
f, axes = plt.subplots(3, 1, sharex=True, figsize=(4.8, 2.8))

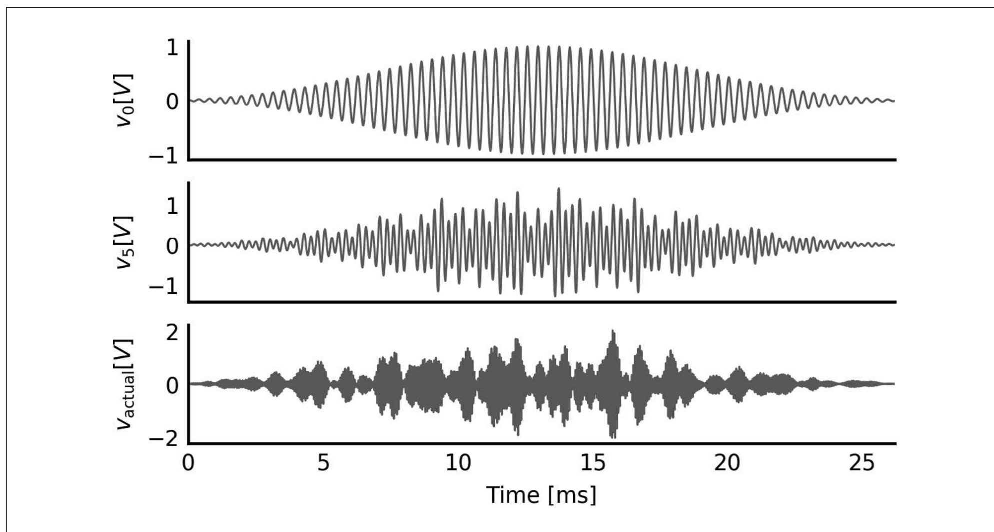
t_ms = t * 1000 # 采样时间，单位为毫秒

w = np.kaiser(N, 6.1) # Kaiser加窗函数，β = 6.1

for n, (signal, label) in enumerate([(v_single, r'$v_0 [V]$'),
                                     (v_sim, r'$v_5 [V]$'),
                                     (v_actual, r'$v_{\mathrm{actual}} [V]$')]):
```

```
    with plt.style.context('style/thinner.mplstyle'):
        axes[n].plot(t_ms, w * signal)
        axes[n].set_ylabel(label)
        axes[n].grid()

axes[2].set_xlim(0, t_ms[-1])
axes[2].set_xlabel('Time [ms]');
```



相应快速傅里叶变换结果的雷达术语为“距离跟踪”。

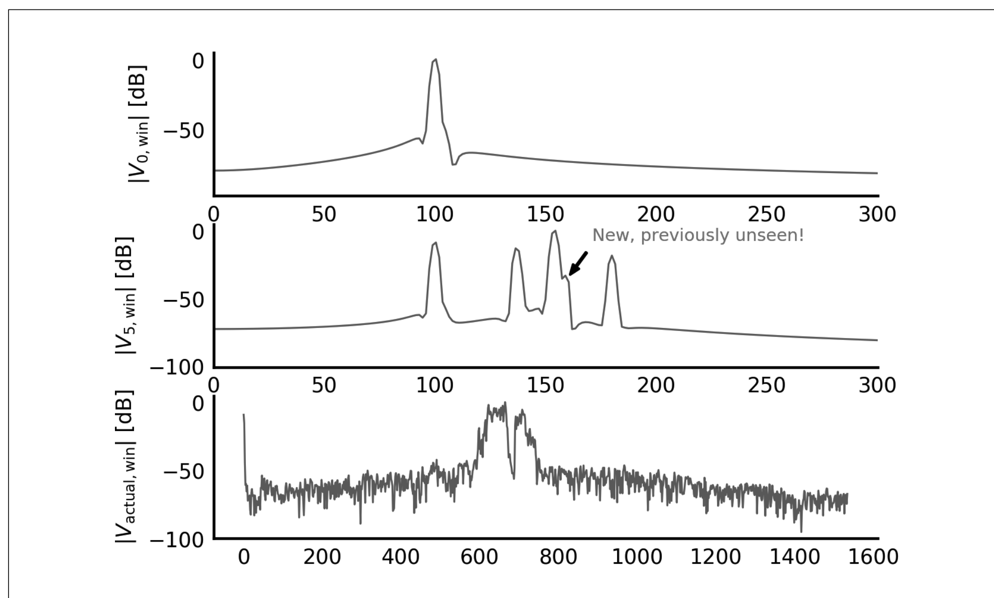
```
V_single_win = np.fft.fft(w * v_single)
V_sim_win = np.fft.fft(w * v_sim)
V_actual_win = np.fft.fft(w * v_actual)

fig, (ax0, ax1,ax2) = plt.subplots(3, 1)

with plt.style.context('style/thinner.mplstyle'):
    log_plot_normalized(rng, V_single_win[:N // 2],
                        r"$|V_{0,\mathrm{win}}|$ [dB]", ax0)
    log_plot_normalized(rng, V_sim_win[:N // 2],
                        r"$|V_{5,\mathrm{win}}|$ [dB]", ax1)
    log_plot_normalized(rng, V_actual_win[:N // 2],
                        r"$|V_\mathrm{actual,win}|$ [dB]", ax2)

ax0.set_xlim(0, 300)  # 修改这些图形的x轴范围,
ax1.set_xlim(0, 300)  # 以便更清楚地看到波峰形状

ax1.annotate("New, previously unseen!", (160, -35), xytext=(10, 15),
             textcoords="offset points", color='red', size='x-small',
             arrowprops=dict(width=0.5, headwidth=3, headlength=4,
                             fc='k', shrink=0.1));
```



可以将这几个图和前面的距离跟踪比较一下。它们显著降低了旁瓣水平，但代价是波峰形状发生了改变，变宽了一些，而且不那么尖锐，由此降低了雷达的分辨率，即雷达分辨两个间隔很近的目标的能力。选择窗口函数时，要在旁瓣水平和分辨率之间进行权衡。尽管如此，从对 V_{sim} 的跟踪结果来看，加窗还是很有效果的，它显著提高了分辨邻近大目标的小目标的能力。

在真实雷达数据的距离跟踪中，加窗同样也会降低旁瓣水平。从两组目标间的缺口深度来看，这一点更加明显。

4.7.3 雷达图像

了解如何分析单一维度的距离跟踪数据后，接下来看看如何分析雷达图像。

数据是由带有抛物线反射面天线的雷达生成的，这种雷达可以在半功率点之间以 2 度的分散角发出高度指向式的横截面为圆形的笔形波束。当垂直照射到一个平面时，雷达会在 60 米距离处照射出一个直径为 2 米的圆形区域。在这个圆形区域外，能量会迅速减少，但还是能够探测到这个区域之外的强回波。

通过改变笔形波束的方位角（左－右位置）和仰角（上－下位置），可以扫描感兴趣的目标区域。当收到反射信号时，可以计算出反射物（被雷达信号击中的物体）的距离，结合当前笔形波束的方位角和仰角，就可以确定这个反射物的三维位置。

一个岩石边坡包含数以千计的反射物。我们可以认为距离箱是沿着崎岖表面与边坡相交的、中心带有雷达信号的大球。在这个距离箱中，相交线上的散射体会产生反射。雷达的波长（发射波在一个震荡周期内的传播距离）大约是 30 毫米。如果散射体之间的距离是 1/4 波长（大约 7.5 毫米）的奇数倍，那么它们的反射信号会因为彼此间的干涉而减弱。如

果散射体之间的距离是 1/2 波长的整数倍，那么它们的反射信号会因为彼此间的干涉而增强。这些反射信号可以组合产生一个明显的强反射区域。这种雷达通过转动天线来扫描小型区域，转动的方位角范围是 20 度，仰角范围是 30 度，每次转动 0.5 度。

现在画出最终雷达数据的轮廓图。来看看不同方向上的数据切片（见图 4-13）。第一个是某个距离上的切片，表示随仰角和方位角变化的回波强度。另外两个切片分别是某个仰角和某个方位角上的切片，可以表示出坡度（见图 4-13 和图 4-14）。可以在方位角图中看到一个露天矿高墙的层叠结构。

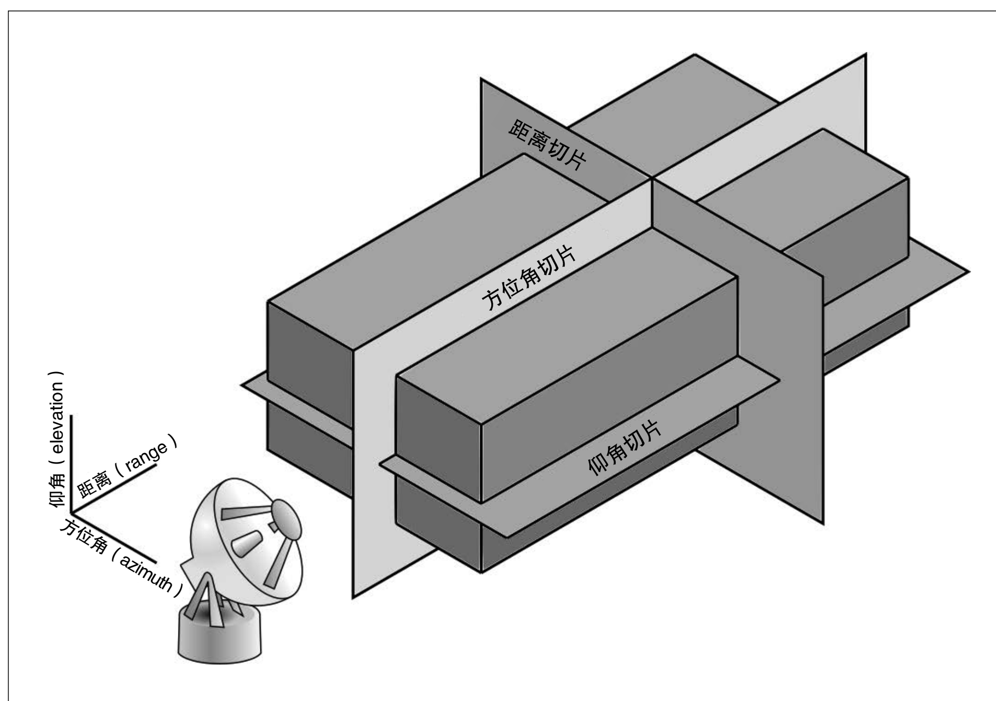


图 4-13：图中显示了对数据卷的方位角切片、仰角切片和距离切片

```
data = np.load('data/radar_scan_1.npz')
scan = data['scan']

# 信号的振幅范围是−2.5V~+2.5V。雷达中的14位模拟数字转换器可以给出−8192~8192的整数值
# 通过乘以(2.5 / 8192)，可以转换回电压

v = scan['samples'] * 2.5 / 8192
win = np.hanning(N + 1)[:-1]

# 对每个测量结果进行快速傅里叶变换
V = np.fft.fft(v * win, axis=2)[::-1, :, :N // 2]

contours = np.arange(-40, 1, 2)

# 忽略MPL布局警告
```

```
import warnings
warnings.filterwarnings('ignore', '.*Axes.*compatible.*tight_layout.*')

f, axes = plt.subplots(2, 2, figsize=(4.8, 4.8), tight_layout=True)

labels = ('Range', 'Azimuth', 'Elevation')

def plot_slice(ax, radar_slice, title, xlabel, ylabel):
    ax.contourf(dB(radar_slice), contours, cmap='magma_r')
    ax.set_title(title)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_facecolor(plt.cm.magma_r(-40))

with plt.style.context('style/thinner.mplstyle'):
    plot_slice(axes[0, 0], V[:, :, 250], 'Range=250', 'Azimuth', 'Elevation')
    plot_slice(axes[0, 1], V[:, 3, :], 'Azimuth=3', 'Range', 'Elevation')
    plot_slice(axes[1, 0], V[6, :, :].T, 'Elevation=6', 'Azimuth', 'Range')
    axes[1, 1].axis('off')
```

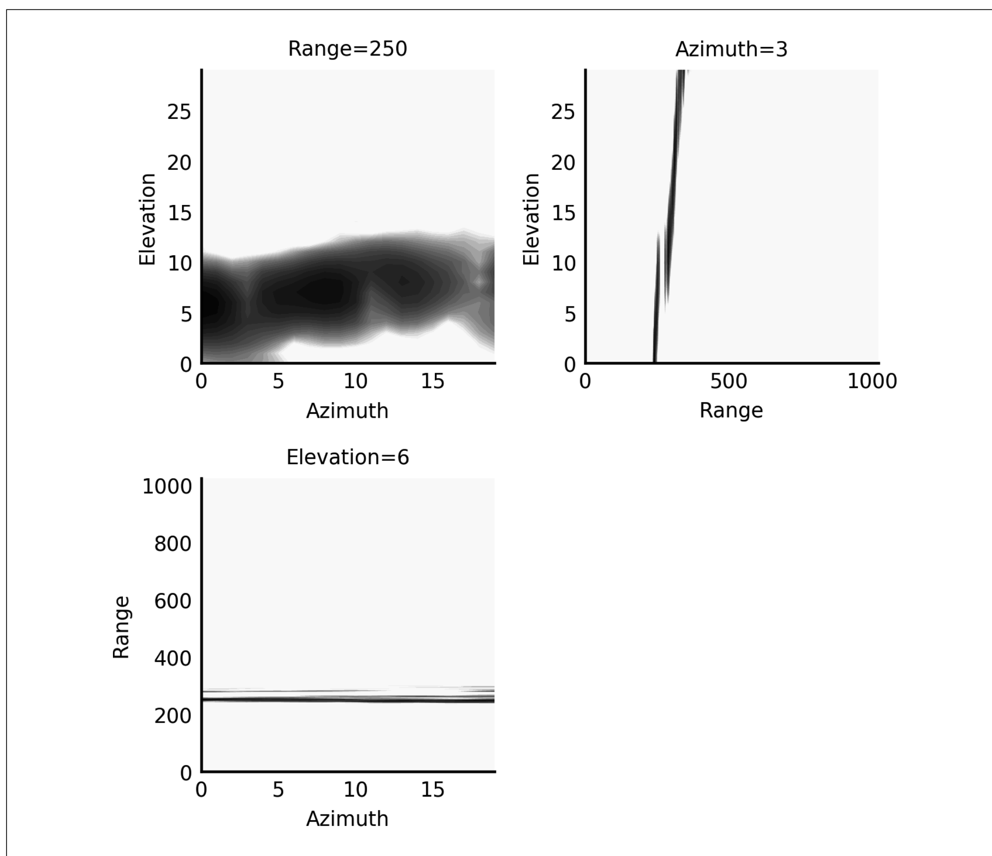


图 4-14：不同坐标轴上的距离跟踪轮廓图（见图 4-13）

三维可视化

还可以实现三维数据的可视化（见图 4-15）。

首先计算出距离方向上的 argmax（最大值索引），它可以表示出雷达波束击中岩石边坡的距离。然后将每个 argmax 索引转换为三维坐标（仰角 – 方位角 – 距离）。

```
r = np.argmax(V, axis=2)

el, az = np.meshgrid(*[np.arange(s) for s in r.shape], indexing='ij')

axis_labels = ['Elevation', 'Azimuth', 'Range']
coords = np.column_stack((el.flat, az.flat, r.flat))
```

通过使用这些坐标，可以将它们投影到方位角 – 仰角平面上（丢弃距离坐标），并用德洛内方法进行曲面细分。这种曲面细分会返回一组能表示三角形（或单纯形）的坐标。但严格来说，这种三角形是用投影坐标定义的，用初始坐标进行重构，将距离分量加回来。

```
from scipy import spatial

d = spatial.Delaunay(coords[:, :2])
simplexes = coords[d.vertices]
```

为了便于显示，我们将距离轴放在最前面。

```
coords = np.roll(coords, shift=-1, axis=1)
axis_labels = np.roll(axis_labels, shift=-1)
```

下面可以使用 Matplotlib 的 `trisurf` 函数将结果可视化。

```
# 这个导入语句用于初始化Matplotlib的三维机制
from mpl_toolkits.mplot3d import Axes3D

# 设置三维轴
f, ax = plt.subplots(1, 1, figsize=(4.8, 4.8),
                     subplot_kw=dict(projection='3d'))

with plt.style.context('style/thinner.mplstyle'):
    ax.plot_trisurf(*coords.T, triangles=d.vertices, cmap='magma_r')

    ax.set_xlabel(axis_labels[0])
    ax.set_ylabel(axis_labels[1])
    ax.set_zlabel(axis_labels[2], labelpad=-3)
    ax.set_xticks([0, 5, 10, 15])

# 调整相机位置以匹配上图
ax.view_init(azim=-50);
```

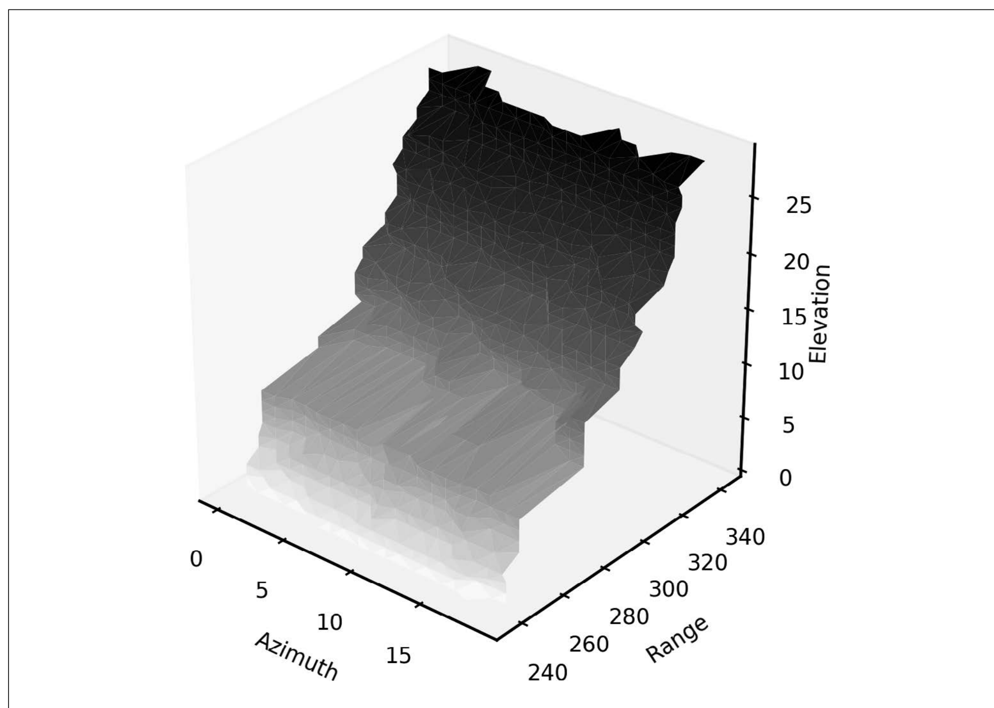


图 4-15：岩石边坡位置估计的三维可视化

4.7.4　快速傅里叶变换的进一步应用

以上示例只展示了快速傅里叶变换在雷达系统中的一种应用。快速傅里叶变换还有很多其他用途，比如运动（多普勒）测量和目标识别。快速傅里叶变换无处不在，从 MRI 到统计学，都可以看到其应用。学习完本章介绍的基本技术后，你应该可以熟练地掌握并使用这种技术。

4.7.5　更多阅读

关于傅里叶变换的图书如下所示。

- PAPOULIS A. The Fourier integral and its applications [M]. New York: McGraw-Hill, 1960.
- BRACEWELL R A. The Fourier transform and its applications [M]. New York: McGraw-Hill, 1986.

关于雷达信号处理的图书如下所示。

- RICHARDS M A, SCHEER J A, HOLM W A. Principles of modern radar: basic principles [G]. Raleigh, NC: SciTech, 2010.
- RICHARDS M A. Fundamentals of radar signal processing [M]. New York: McGraw-Hill, 2014.

4.7.6　练习：图像卷积

快速傅里叶变换经常用于加速图像卷积（卷积是滑动滤波器的一种应用）。可以通过 NumPy 的 `np.convolve` 函数或者 `np.fft.fft2` 函数使用 `np.ones((5, 5))` 对图像进行卷积。确认它们的结果是相同的。

以下是一些提示。

- `x` 和 `y` 的卷积等价于 `ifft2(X * Y)`，其中 `X` 和 `Y` 分别是 `x` 和 `y` 的快速傅里叶变换。
- 为了使 `X` 和 `Y` 相乘，它们必须是同样大小。对 `x` 和 `y` 进行快速傅里叶变换前，`np.pad` 用 0（向右侧和下方）扩展 `x` 和 `y`。
- 你可能会看到一些边缘效应。可以增加补 0 的宽度来消除这种效应，使 `x` 和 `y` 的维度都是 `shape(x) + shape(y) - 1`。

参见附录 A.5 节。

第5章

用稀疏坐标矩阵实现列联表

> 我喜欢稀疏性。它有一种特质，可以通过最少的情感创造出一种非常直接的冲击，并且独一无二。我可能将一直使用这种原则来工作，我也不知道为什么。
>
> ——Britt Daniel，Spoon 乐队主唱

现实世界中的很多矩阵都是**稀疏的**，这意味着其中的多数元素为零。

用 NumPy 数组处理稀疏矩阵会浪费大量时间和精力，因为要用很多元素乘以零。相反，我们可以用 SciPy 的 `sparse` 模块来有效地处理稀疏矩阵，它只检查那些非零的元素。除了解决这些“典型的”稀疏矩阵问题，`sparse` 还可以用于解决那些与稀疏矩阵并非明显相关的问题。

图像分割的比较就是这种问题的一个示例。（复习一下第 3 章中图像分割的定义。）

本章的开场示例代码中使用了两次稀疏矩阵。首先，用 Andreas Mueller 推荐的代码计算一个**列联矩阵**（contingency matrix），这个矩阵为两个分区间的标签对应提供计数。然后，根据 Jaime Fernández del Río 和 Warren Weckesser 的建议，用列联矩阵计算**信息变异**，它可以用来衡量分区间的差异。

```
def variation_of_information(x, y):
    # 计算列联矩阵，即联合概率矩阵
    n = x.size
    Pxy = sparse.coo_matrix((np.full(n, 1/n), (x.ravel(), y.ravel())),
                            dtype=float).tocsr()

    # 计算边际概率，转换为一维数组
    px = np.ravel(Pxy.sum(axis=1))
    py = np.ravel(Pxy.sum(axis=0))

    # 先用稀疏矩阵线性代数计算VI，计算反对角矩阵
```

```
    Px_inv = sparse.diags(invert_nonzero(px))
    Py_inv = sparse.diags(invert_nonzero(py))

    # 然后计算出熵
    hygx = px @ xlog1x(Px_inv @ Pxy).sum(axis=1)
    hxgy = xlog1x(Pxy @ Py_inv).sum(axis=0) @ py

    # 返回它们的和
    return float(hygx + hxgy)
```

> **Python 3.5 增强特性**
>
> 上述代码中的 @ 符号表示 2015 年引入 Python 3.5 的**矩阵乘法**（matrix multiplication）操作符。对于科学编程者来说，这是使用 Python 3 的最令人信服的理由之一：它使得我们可以用与初始数学公式非常相似的代码实现线性代数算法。比较以下两种代码，先是第一种。
>
> ```
> hygx = px @ xlog1x(Px_inv @ Pxy).sum(axis=1)
> ```
>
> 然后是第二种，即 Python 2 中的等价代码。
>
> ```
> hygx = px.dot(xlog1x(Px_inv.dot(Pxy)).sum(axis=1))
> ```
>
> 使用 @ 操作符的代码与数学表示非常相似，从而可以避免程序错误，写出更易读的代码。
>
> 实际上，SciPy 开发者早在引入 @ 操作符之前就意识到了这一点，而且已经在输入为 SciPy 矩阵时修改了 * 操作符的含义。Python 2.7 中就可以写出像上面那样既优雅又易于阅读的代码。
>
> ```
> hygx = -px * xlog(Px_inv * Pxy).sum(axis=1)
> ```
>
> 但是这样做有一个巨大的隐患：px 或 Px_inv 可能是 SciPy 矩阵，也可能不是 SciPy 矩阵，不同的情况下这行代码的作用完全不同！如果 Px_inv 和 Pxy 是 NumPy 数组，那么 * 表示元素级别的乘法；但如果 Px_inv 和 Pxy 是 SciPy 矩阵，那么 * 表示矩阵乘积！可以想象，这将会导致很多错误，因此很多 SciPy 社区禁止这种做法，宁可使用丑陋但没有歧义的 .dot 方法。
>
> Python 3.5 引入 @ 操作符是一种两全其美的做法！

5.1 列联表

我们先从简单工作开始，然后逐渐转到图像分割。

假设你刚在电子邮件创业公司 Spam-o-matic 开始自己的职业生涯。你的任务是建立一个垃圾邮件检测器，用数值表示探测器的结果：0 表示非垃圾邮件，1 表示垃圾邮件。

如果你对一个包含 10 封邮件的集合进行分类，可以得到一个**预测**向量。

```
import numpy as np
pred = np.array([0, 1, 0, 0, 1, 1, 1, 0, 1, 1])
```

你可以与包含**真实结果**的向量比较一下，真实结果是通过手工检查每封邮件得到的一个正确分类。

```
gt = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
```

眼下，分类对计算机来说还是比较困难的，因此 `pred` 和 `gt` 中的值不能完全匹配。在 `pred` 和 `gt` 都是 0 的位置，如果预测器正确地识别出一封邮件不是垃圾邮件，那么这种情况称为**真阴性**（true negative）。反之，如果预测器在都是 1 的位置正确地识别出一封垃圾邮件，那就是**真阳性**（true positive）。

错误类型主要有两种。如果将一封垃圾邮件（`gt` 值为 1）放到了用户的收件箱（`pred` 值为 0），那么我们就犯了**假阴性**（false negative）错误。如果将一封合法的邮件（`gt` 值为 0）预测为垃圾邮件（`pred` 值为 1），那么我们就犯了**假阳性**（false positive）错误。（有一次，我们科学研究所主任的一封邮件就跑到了我的垃圾邮件文件夹。为什么呢？他发的博士后演讲比赛通知开头是“你可以赢得 500 美元”！）

如果想要测量工作效果，就必须用**列联矩阵**考虑以上所说的几种错误。（列联矩阵又称为混淆矩阵，这个名称非常贴切。）为了得到列联矩阵，我们将预测标签放在行上，真实结果标签放在列上，然后计算出它们能够互相对应的次数。例如，因为有 4 个真阳性结果（`pred` 和 `gt` 均为 1），所以矩阵在 `(1, 1)` 处的值是 4。

通常来说：

$$C_{ij} = \sum_k \mathbb{I}(p_k = i)\mathbb{I}(g_k = j)$$

以下代码是这个公式的一种直观但低效的实现。

```
def confusion_matrix(pred, gt):
    cont = np.zeros((2, 2))
    for i in [0, 1]:
        for j in [0, 1]:
            cont[i, j] = np.sum((pred == i) & (gt == j))
    return cont
```

我们可以检查它能否给出正确的计数结果。

```
confusion_matrix(pred, gt)

array([[ 3., 1.],
       [ 2., 4.]])
```

5.1.1 练习：混淆矩阵的计算复杂度

为什么说上述那段代码低效？

参见附录 A.6 节。

5.1.2 练习：计算混淆矩阵的另一种方法

计算混淆矩阵的另一种方法只需要遍历一次 `pred` 和 `gt`。

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    # 你的代码在这里
    return cont
```

参见附录 A.7 节。

我们可以使这个示例更通用一点。垃圾邮件和非垃圾邮件的分类太简单，可以进一步将邮件分为垃圾邮件、新闻、广告促销、邮件列表和个人邮件。这样就共有 5 个类别，标号为 0~4，混淆矩阵为 5 × 5，对角线上的值表示匹配计数，非对角线上的值表示错误计数。

`confusion_matrix` 函数的定义不能很好地扩展为这种更大的矩阵，因为必须遍历预测结果数组和实际结果数组 25 次。如果加入更多的电子邮件分类（如社交媒体通知），那么问题就更大了。

5.1.3 练习：多类混淆矩阵

像前面一样，编写一个函数在一次遍历中计算混淆矩阵，但不是假设有 2 个分类，而是通过输入确定分类个数。

```
def general_confusion_matrix(pred, gt):
    n_classes = None # 用某个有意义的对象替换'None'
    # 你的代码在这里
    return cont
```

你的一次遍历解决方案应该根据类的数目很好地扩展，但是，由于 `for` 循环是在 Python 解释器中运行的，当邮件数量过多时，程序的速度会非常慢。此外，因为有些类彼此之间非常容易混淆，所以矩阵将是非常**稀疏的**，其中很多元素为 0。实际上，当分类数量增加时，由于列联矩阵的 0 值元素占用了大量内存空间，内存的浪费就更加严重了。因此，可以使用 SciPy 的 `sparse` 模块，它包含了能够有效处理稀疏矩阵的对象。

5.2 scipy.sparse数据格式

第 1 章中介绍了 NumPy 数组的内部数据格式。我们希望你认同，它是保存 *N* 维数组数据的非常直观的格式，并且从某种程度上是必需的。对于稀疏矩阵来说，确实可以有多种数据格式，而且“正确的”格式要依具体问题而定。我们将介绍两种最常用的格式，如果想知道完整的格式列表，参见本章后面用于比较的一个表格，或者 `scipy.sparse` 的在线文档。

5.2.1 COO格式

最简单直观的可能就是坐标系格式，或称 COO 格式。它用 3 个一维数组来表示一个二维矩阵 ***A***。每个数组的长度都等于 ***A*** 中非零元素的数量，它们共同组成了一个包括所有非零元素坐标 (`i,j,值`) 的列表。

- `row` 数组和 `col` 数组分别表示行索引和列索引，它们共同确定了每个非零元素的位置。
- `data` 数组确定了每个位置的**值**。

数组中没有被数对 (row, col) 表示的所有元素都默认为 0。这样效率就高多了！因此，需要表示这个矩阵。

```
s = np.array([[ 4,  0, 3],
              [ 0, 32, 0]], dtype=float)
```

可以使用以下代码。

```
from scipy import sparse

data = np.array([4, 3, 32], dtype=float)
row = np.array([0, 0, 1])
col = np.array([0, 2, 1])

s_coo = sparse.coo_matrix((data, (row, col)))
```

scipy.sparse 中的 .toarray() 方法可以将稀疏矩阵格式还原为稀疏数据的 NumPy 数组表示形式。可以用这个方法检查是否正确创建了 s_coo。

```
s_coo.toarray()

array([[ 4.,   0.,  3.],
       [ 0.,  32.,  0.]])
```

同样也可以使用 .A **属性**，它就像一个特征，但实际上是运行一个函数。.A 是一个特别危险的属性，因为其中隐藏了开销巨大的操作：稀疏矩阵 NumPy 数组版本耗费的资源要比稀疏矩阵本身大好几个数量级，只需 3 次按键，就可能让计算机瘫痪！

```
s_coo.A

array([[ 4.,   0.,  3.],
       [ 0.,  32.,  0.]])
```

只要不影响可读性，本章或其他地方都推荐使用 toarray() 方法，因为它能更清楚地标识出潜在的代价昂贵的操作。但如果 .A 的简洁性可以使代码更易读（例如，尝试实现数学等式的序列时），也可以使用 .A。

5.2.2　练习：COO表示

写出以下矩阵的 COO 表示：

```
s2 = np.array([[0, 0, 6, 0, 0],
               [1, 2, 0, 4, 5],
               [0, 1, 0, 0, 0],
               [9, 0, 0, 0, 0],
               [0, 0, 0, 6, 7]])
```

遗憾的是，尽管简单直观，但 COO 格式的优化程度还不够，它没有使用最少的内存或在计算时尽快遍历数组。（回忆一下第 1 章，**数据局部性**对于提高计算效率非常重要！）然而，你可以仔细查看一下上面的 COO 表示，以找出冗余信息。你能在 1 秒内找出重复元素吗？

5.2.3 稀疏行压缩格式

如果用 COO 一行行地枚举非零元素，而不是按任意顺序枚举（COO 格式允许这样做），那么就可以在 `row` 数组中得到很多连续的、重复的值。可以通过引用 `col` 中下一行开始的索引来压缩这些值，这样可以避免重复使用行索引。这就是**稀疏行压缩**（CSR，compressed sparse row）格式的基本思想。

我们用以上示例来说明。在 CSR 格式中，`col` 和 `data` 数组不变（但 `col` 被重新命名为 `indices`）。但 `row` 数组不再表示行索引，而是表示每一行在 `col` 数组中从哪里开始，而且被重新命名为 `indptr`，含义是“索引指针”。

先看一下 COO 格式中的 `row` 和 `col`，暂且忽略 `data`。

```
row = [0, 1, 1, 1, 1, 2, 3, 4, 4]
col = [2, 0, 1, 3, 4, 1, 0, 3, 4]
```

每当 `row` 中索引值发生变化时，就会开始一个新行。第 0 行在索引 0 处开始，第 1 行在索引 1 处开始，但第 2 行在 `row` 中第一次出现“2”的地方，也就是索引 5 处开始。对于第 3 行和第 4 行，索引位置每次增加 1，就是第 6 和第 7 个索引。还要加上最后一个表示矩阵结束的索引，它是矩阵中非零值的总数（9）。因此：

```
indptr = [0, 1, 5, 6, 7, 9]
```

接下来用这些简单的数组来建立 SciPy 中的 CSR 矩阵。前面定义了一个 NumPy 数组 `s2`，通过比较其 COO 表示和 CSR 表示的 `.A` 输出，可以检验我们的工作是否正确。

```
data = np.array([6, 1, 2, 4, 5, 1, 9, 6, 7])

coo = sparse.coo_matrix((data, (row, col)))
csr = sparse.csr_matrix((data, col, indptr))

print('The COO and CSR arrays are equal: ',
      np.all(coo.A == csr.A))
print('The CSR and NumPy arrays are equal: ',
      np.all(s2 == csr.A))

The COO and CSR arrays are equal: True
The CSR and NumPy arrays are equal: True
```

这种保存大型稀疏矩阵并在其上进行计算的能力强大到不可思议，可以应用于多个领域。

例如，可以认为整个互联网是一个大型的、稀疏的 $N \times N$ 矩阵。其中的每一项 X_{ij} 表示网页 i 是否链接到网页 j。对这个矩阵进行标准化并求出主特征向量，就可以得到所谓的 PageRank——Google 用来对搜索结果进行排序的一种数值。（下一章将对此做更多介绍。）

再看一个示例，可以将人脑表示成一个 $m \times m$ 的大型图结构，其中有 m 个可以用 MRI 扫描仪检测其活动的节点（位置）。检测一段时间后，可以计算出它们之间的相关性并保存到一个矩阵 $\boldsymbol{C}_{ij}$ 中。对这个矩阵进行阈值化，可以得到一个包含 1 和 0 的稀疏矩阵。与这个矩阵的第二小的特征值对应的特征向量可以将 m 个大脑区域分成几个子组，事实证明，它们经常与大脑的功能区域相关！[1]

注 1：NEWMAN M E J. Modularity and community structure in networks [J]. PNAS, 2006, 103(23): 8577-8582.

	BSR矩阵	COO矩阵	CSC矩阵	CSR矩阵	DIA矩阵	DOK矩阵	LIL矩阵
全称	块稀疏行矩阵	坐标矩阵	稀疏列压缩矩阵	稀疏行压缩矩阵	对角矩阵	字典键矩阵	基于行的链接列表矩阵
说明	与 CSR 相似	仅用于构造稀疏矩阵，然后转换为 CSC 或 CSR 做进一步处理	—	—	—	用于增量构造稀疏矩阵	用于增量构造稀疏矩阵
应用场景	存储稠密的子矩阵；常用于离散问题的数值分析，如有限元、差分方程	构造稀疏矩阵的快速、直接的方法；构造矩阵时，同样的坐标可以相加，这在有限元分析中很有用	算术操作（支持加、减、乘、除、矩阵乘方）；有效的列切片；快速的矩阵－向量相乘（CSR、BSR 可能更快，依具体问题而定）	算术操作；有效的行切片；快速的矩阵－向量相乘	算术操作	转换为稀疏结构的成本很低；算术操作；快速访问单个元素；有效地转换为 COO（不允许复制）	转换为稀疏结构的成本很低；灵活的切片操作
缺点	—	不支持算术运算；不支持切片	行切片速度慢（见 CSR）；转换为稀疏结构的成本高（见 LIL、DOK）	列切片速度慢（见 CSC）；转换为稀疏结构的成本高（见 LIL、DOK）	转换为稀疏结构时，只在对角线上有值	算术运算成本高；矩阵－向量相乘速度慢	算术运算成本高；列切片速度慢；矩阵－向量相乘速度慢

5.3 稀疏矩阵应用：图像转换

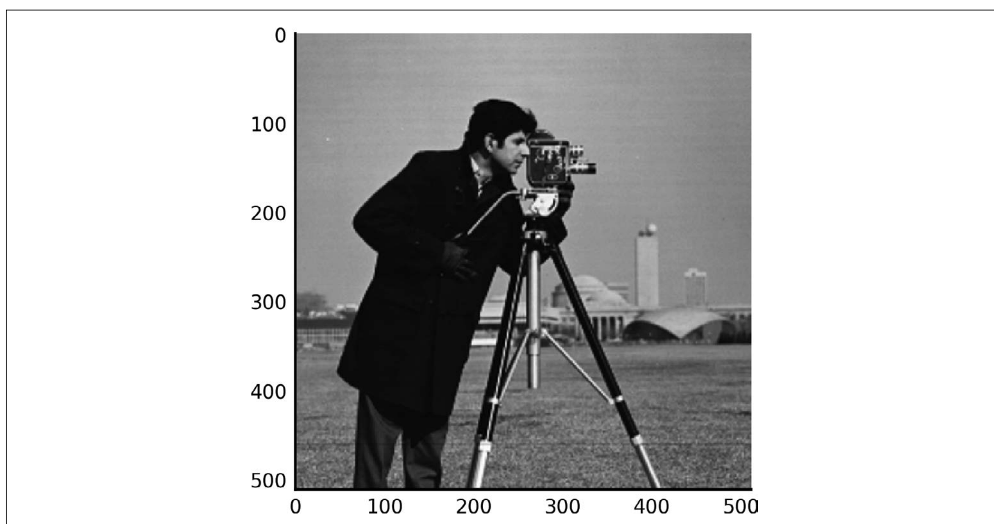
一些库已经包含了有效的图像转换（旋转与变形）算法，如 scikit-image 和 SciPy。但是，假设你是 NumPy 航天事务局的头儿，必须对一个新发射的木星轨道飞行器传回来的成千上万张图像进行旋转，那么该怎么办呢？

在这种情况下，你必须让计算机使出吃奶的力气。事实证明，如果**重复应用**同一种转换，那么效果要远远好于 SciPy 的 `ndimage` 模块中那些经过优化的 C 代码。

以下是一张来自 scikit-image 的测试图片，其中是一个正在照相的男人，我们用这张图片作为示例数据。

```
# 使图形显示在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')

from skimage import data
image = data.camera()
plt.imshow(image);
```



作为一项测试，将这张图片旋转 30 度。先定义转换矩阵 $\boldsymbol{H}$，用它乘以输入图像的坐标 $[r, c, 1]$ 可以得到相应的输出坐标 $[r', c', 1]$。（注意：我们使用的是齐次坐标，它的后面有一个 1，这样在定义线性转换时就更具灵活性。）

```
angle = 30
c = np.cos(np.deg2rad(angle))
s = np.sin(np.deg2rad(angle))

H = np.array([[c, -s,  0],
              [s,  c,  0],
              [0,  0,  1]])
```

可以用 $\boldsymbol{H}$ 乘以点 (1, 0) 来检验效果。绕着原点 (0, 0) 逆时针旋转 30 度可以得到点 ($\sqrt{3}$ /2, 1/2)。

```
point = np.array([1, 0, 1])
print(np.sqrt(3) / 2)
print(H @ point)

0.866025403784
[ 0.8660254  0.5        1.        ]
```

同理，3 次 30 度旋转可以将点转到纵轴上，得到点 (0, 1)。可以看到，除了一些浮点数近似误差，确实是这样的。

```
print(H @ H @ H @ point)

[  2.77555756e-16   1.00000000e+00   1.00000000e+00]
```

接下来要建立一个函数来定义“稀疏操作符”。稀疏操作符的目的是得出输出图像中的所有像素，先确定它们在输入图像中的位置，再用合适的（双线性）插值方法（见图 5-1）计算出它们的值。它用矩阵乘法来完成这个任务，因此速度特别快。

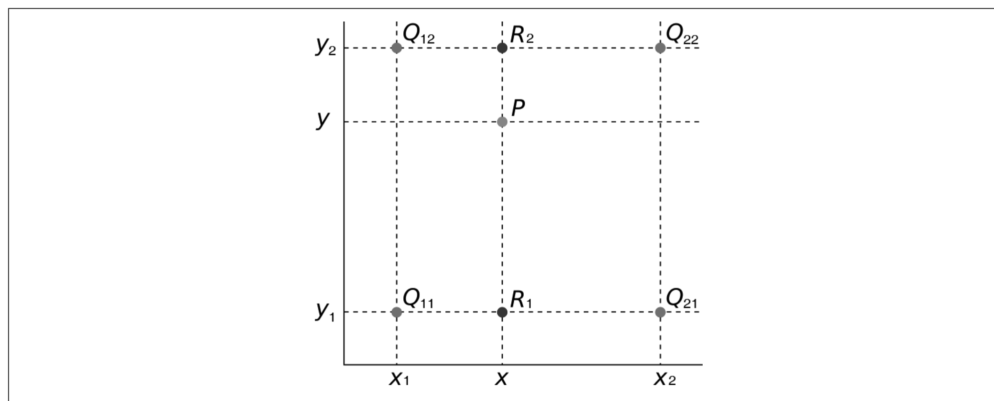


图 5-1：双线性插值示意图——P 点的值是根据 Q_{11}、Q_{12}、Q_{21}、Q_{22} 的加权和估计出来的

下面来看一下建立稀疏操作符的函数。

```
from itertools import product

def homography(tf, image_shape):
    """Represent homographic transformation & interpolation as linear operator.

    Parameters
    ----------
    tf : (3, 3) ndarray
        Transformation matrix.
    image_shape : (M, N)
        Shape of input gray image.

    Returns
    -------
    A : (M * N, M * N) sparse matrix
```

```
        Linear-operator representing transformation + bilinear interpolation.

    """
    # 逆矩阵，能告诉我们与每个输出像素所对应的输入像素的位置。
    H = np.linalg.inv(tf)

    m, n = image_shape

    # 我们要构建一个COO矩阵，常称为IJK矩阵，需要行坐标（I）、列坐标（J）和值（K）。
    row, col, values = [], [], []

    # 对输出图像中的每个像素……
    for sparse_op_row, (out_row, out_col) in \
            enumerate(product(range(m), range(n))):

        # 计算出它们在输入图像中的位置
        in_row, in_col, in_abs = H @ [out_row, out_col, 1]
        in_row /= in_abs
        in_col /= in_abs

        # 如果坐标跑到了初始图像外，则忽略该坐标，这个位置的值就是0
        if (not 0 <= in_row < m - 1 or
                not 0 <= in_col < n - 1):
            continue

        # 我们要找出输出像素周围的4个像素，通过对它们的值进行插值，
        # 得出对输出像素的精确估计。从左上角开始，注意其余的点在每个方向上都在1个
        # 单位的距离外
        top = int(np.floor(in_row))
        left = int(np.floor(in_col))

        # 计算输出像素的位置，映射到输入图像，在4个所选像素之间
        t = in_row - top
        u = in_col - left

        # 稀疏操作符矩阵的当前行由输出像素坐标的笛卡儿积顺序确定，
        # 保存在sparse_op_row中。要算出对应4个列的周围4个输入像素的加权平均，
        # 因此需要将行索引重复4次
        row.extend([sparse_op_row] * 4)

        # 实际的权重是根据双线性插值算法计算出来的
        sparse_op_col = np.ravel_multi_index(
                ([top,  top,      top + 1, top + 1 ],
                 [left, left + 1, left,    left + 1]), dims=(m, n))
        col.extend(sparse_op_col)
        values.extend([(1-t) * (1-u), (1-t) * u, t * (1-u), t * u])

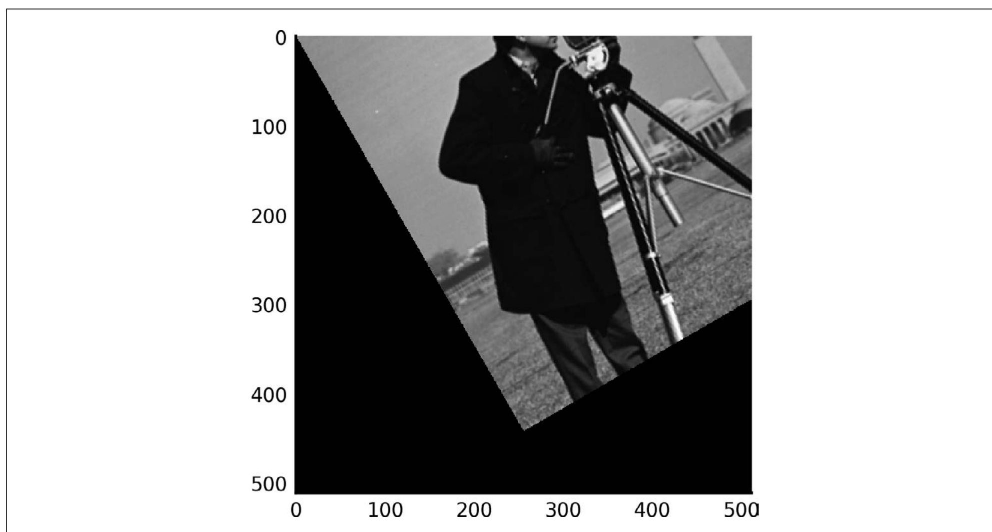
    operator = sparse.coo_matrix((values, (row, col)),
                                 shape=(m*n, m*n)).tocsr()
    return operator
```

像下面这样使用稀疏操作符。

```
def apply_transform(image, tf):
    return (tf @ image.flat).reshape(image.shape)
```

试一下！

```
tf = homography(H, image.shape)
out = apply_transform(image, tf)
plt.imshow(out);
```



这就是旋转的效果！

练习：图像旋转

旋转是围绕着坐标原点 (0, 0) 进行的，你能绕着图像中心进行旋转吗？

提示：图像翻译（即上下或左右移动图像）的转换矩阵如下所示。

$$H_{tr} = \begin{bmatrix} 1 & 0 & t_r \\ 0 & 1 & t_c \\ 0 & 0 & 1 \end{bmatrix}$$

以上公式可以将图像向下移动 t_r 个像素，向右移动 t_c 个像素。

如前所述，这种用于图像变换的线性稀疏操作符方法速度非常快。接下来测量一下，与 ndimage 相比，其速度到底有多快。为了公平起见，我们要告诉 ndimage 使用 order = 1 的线性插值，并通过 reshape = False 忽略那些跑到初始形状外的像素。

```
%timeit apply_transform(image, tf)

100 loops, average of 7: 3.35 ms +- 270 μs per loop (using standard deviation)

from scipy import ndimage as ndi
%timeit ndi.rotate(image, 30, reshape=False, order=1)

100 loops, average of 7: 19.7 ms +- 988 μs per loop (using standard deviation)
```

可以看出，它在计算机上的速度大约快了 10 倍。尽管这只是一个旋转的例子，但完全可以进行更复杂的变形操作，比如修正成像过程中的扭曲镜头，或者给人脸加上一个滑稽的表情。一旦完成转换计算，重复应用这种转换的速度就会非常快，这都要归功于稀疏矩阵代数。

介绍完 SciPy 稀疏矩阵的“标准”应用后，接下来将介绍促使我们撰写本章的非传统应用。

5.4 回到列联表

你可能还记得，我们试图用 SciPy 稀疏矩阵格式快速建立一个稀疏的联合概率矩阵。我们知道 COO 格式可以将稀疏数据保存为 3 个数组，其中分别包含非零元素的行坐标、列坐标及其值。我们可以用 COO 的一些已知特性快速得到需要的矩阵。

查看以下数据。

```
row = [0, 0, 2]
col = [1, 1, 2]
dat = [5, 7, 1]
S = sparse.coo_matrix((dat, (row, col)))
```

注意，(行 , 列) 位置为 (0, 1) 的元素出现了两次：第一次是 5，然后是 7。那么矩阵在 (0, 1) 处的值到底是什么呢？可能是前一个值，也可能是后一个值，但实际上是两个值的**和**。

```
print(S.toarray())

[[ 0 12  0]
 [ 0  0  0]
 [ 0  0  1]]
```

因此，COO 格式会累加重复的元素。这正是构建列联矩阵所需要的！实际上，我们的任务已经基本完成了：可以将行设为 `pred`、列设为 `gt`、值设为 `1`。在矩阵中的 (i, j) 位置，`pred` 中标记为 i 和 `gt` 中标记为 j 的 1 会累加起来，并计算出累加的次数！下面来试一下。

```
from scipy import sparse

def confusion_matrix(pred, gt):
    cont = sparse.coo_matrix((np.ones(pred.size), (pred, gt)))
    return cont
```

我们用 `.toarray` 方法看一个小示例。

```
cont = confusion_matrix(pred, gt)
print(cont)

  (0, 0)    1.0
  (1, 0)    1.0
  (0, 0)    1.0
  (0, 0)    1.0
  (1, 0)    1.0
  (1, 1)    1.0
  (1, 1)    1.0
  (0, 1)    1.0
```

```
  (1, 1)    1.0
  (1, 1)    1.0

print(cont.toarray())

[[ 3.  1.]
 [ 2.  4.]]
```

果真有效！

练习：减少内存占用

第 1 章中介绍过，NumPy 的内置工具可以用**广播机制**重复数组。如何才能减少列联矩阵计算所需的内存占用？

提示：查阅 `np.broadcast_to` 函数的文档。

5.5　图像分割中的列联表

可以用与以上分类问题相同的思路来处理图像分割：每个**像素**的分区标签都是对该像素所属**类**的一个**预测**。NumPy 数组可以让我们轻松地完成这个任务，因为它们的 `.ravel()` 方法可以返回基础数据的一维视图。

举个例子，以下是对 3 × 3 的小图片的一种分割。

```
seg = np.array([[1, 1, 2],
                [1, 2, 2],
                [3, 3, 3]], dtype=int)
```

以下是真实结果，也就是这个图像的正确分割方式。

```
gt = np.array([[1, 1, 1],
               [1, 1, 1],
               [2, 2, 2]], dtype=int)
```

我们可以认为这两个结果就是分类问题，就像前面的分类问题一样。每个像素都是一个不同的预测。

```
print(seg.ravel())
print(gt.ravel())

[1 1 2 1 2 2 3 3 3]
[1 1 1 1 1 1 2 2 2]
```

和前面的例子类似，用以下代码得出列联矩阵。

```
cont = sparse.coo_matrix((np.ones(seg.size),
                          (seg.ravel(), gt.ravel())))
print(cont)

  (1, 1)    1.0
  (1, 1)    1.0
  (2, 1)    1.0
```

```
  (1, 1)    1.0
  (2, 1)    1.0
  (2, 1)    1.0
  (3, 2)    1.0
  (3, 2)    1.0
  (3, 2)    1.0
```

有些索引出现了不止一次，但我们可以用 COO 格式的可加性来确认这就是我们需要的矩阵。

```
print(cont.toarray())

[[ 0.  0.  0.]
 [ 0.  3.  0.]
 [ 0.  3.  0.]
 [ 0.  0.  3.]]
```

如何转换这个表格才能测量出 `seg` 对 `gt` 的表示效果呢？图像分割是一个很困难的问题，因此，通过比较其输出和手工得出的“真实结果”，可以测量出图像分割算法的效果，这是非常重要的。

但这种比较也不是一个容易的任务。那么如何定义自动分割结果与真实结果的“接近”程度呢？下面介绍一种名为**信息变异**（VI，variation of information）[2] 的方法。它被定义为以下问题的答案：从平均意义上来说，对于一个随机的像素，如果给定了它在一个分区内的分区 ID，那么还需要多少**信息**才能确定它在另一个分区内的 ID 呢？

直观地说，如果两个分区完全相同，那么知道一个分区中的 ID 就可以知道另一个分区中的 ID，无须其他信息。但如果两个分区差异较大，在没有更多信息的情况下，知道一个分区中的 ID 无法确定另一个分区中的 ID。

5.6 信息论简介

为了回答之前给出的问题，需要快速了解一下信息论的基础知识。此处需要节省篇幅，但如果你想了解更多内容，可以查看 Christopher Olah 的博文“Visual Informatican Theory”。

信息的基本单位是**位**（bit），通常表示为 0 或 1，这表示两个选项被选中的概率相等。这很容易理解：如果想要告诉你掷硬币的结果是正面还是反面，那么就需要一个位，它有多种形式：沿着电报线路传来的或长或短的脉冲（摩尔斯电码）、两种颜色的灯光闪烁，或一个只能取 0 或 1 的数值。重要的是，我们**一定**需要一个位，因为掷硬币的结果是随机的。

事实上，我们可以将这个概念扩展为**分数**位，以表示那些随机性**较弱**的事件。比如，假设你需要发射一个表示拉斯维加斯今天是否下雨的信号。乍一看，这也需要一个位：0 表示没有下雨，1 表示下雨。但是，拉斯维加斯下雨是非常罕见的事件。因此，随着时间的推移，我们可以偷个懒，仅仅发射比原来少得多的信息：**偶尔**发射 0 以确定我们的通信没有中断，其他时间都简单地**假设**信号为 0，只在下雨这一罕见事件发生时才发射 1。

注 2：MEILA M. Comparing clusterings—an information based distance [J]. Journal of Multivariate Analysis, May 2007, 98(5): 873-895.

因此，当两个事件发生的概率**不等**时，我们需要**少于**一个位来表示它们。一般来说，对于任意随机变量 X（可能值可以多于两个），我们用以下函数 H 表示它的**熵**。

$$H(X)=\sum_x p_x \log_2\left(\frac{1}{p_x}\right)$$
$$=-\sum_x p_x \log_2(p_x)$$

其中 x 是 X 的可能值，p_x 是 X 取 x 值的概率。

因此，如果用 T 表示掷一次硬币这个事件，那么它可能有两个值，即正面（h）和反面（t），则其熵如下所示：

$$\begin{aligned} H(T) &= p_h \log_2(1/p_h) + p_t \log_2(1/p_t) \\ &= 1/2\log_2(2) + 1/2\log_2(2) \\ &= 1/2 \cdot 1 + 1/2 \cdot 1 \\ &= 1 \end{aligned}$$

从长期来看，拉斯维加斯的某一天下雨的概率大概是 1/6。用 R 表示拉斯维加斯下雨这一事件，可能的取值是下雨（r）和晴天（s），则其熵如下所示：

$$\begin{aligned} H(R) &= p_r \log_2(1/p_r) + p_s \log_2(1/p_s) \\ &= 1/6\log_2(6) + 5/6\log_2(6/5) \\ &\approx 0.65 \text{ bits} \end{aligned}$$

条件熵是一种特殊的熵，它是**假设**你知道变量的一些其他相关信息时变量的熵。例如，在已知月份的情况下，下雨这一事件的熵是多少？此时可以表示如下：

$$H(R|M)=\sum_{m=1}^{12} p(m)H(R|M=m)$$

并且：

$$\begin{aligned} H(R|M=m) &= p_{r|m}\log_2\left(\frac{1}{p_{r|m}}\right)+p_{s|m}\log_2\left(\frac{1}{p_{s|m}}\right) \\ &= \frac{p_{rm}}{p_m}\log_2\left(\frac{p_m}{p_{rm}}\right)+\frac{p_{sm}}{p_m}\log_2\left(\frac{p_m}{p_{sm}}\right) \\ &= -\frac{p_{rm}}{p_m}\log_2\left(\frac{p_{rm}}{p_m}\right)-\frac{p_{sm}}{p_m}\log_2\left(\frac{p_{sm}}{p_m}\right) \end{aligned}$$

现在你已经具备了理解信息变异所需的全部信息论基础。在以上示例中，事件是日期，而且它们有两个属性。

- `rain/shine`（下雨 / 晴天）
- `month`（月份）

就像在分类示例中那样，通过观测大量日期，我们可以建立一个**列联矩阵**，表示某个日期的月份以及这一天是否下雨。我们不用跑去拉斯维加斯完成这个任务（尽管这非常有趣），只需使用来自 WeatherSpark 网站的以下历史表格。

月份	*P*(下雨)	*P*(晴天)
1	0.25	0.75
2	0.27	0.73
3	0.24	0.76
4	0.18	0.82
5	0.14	0.86
6	0.11	0.89
7	0.07	0.93
8	0.08	0.92
9	0.10	0.90
10	0.15	0.85
11	0.18	0.82
12	0.23	0.77

给定 `month` 后，`rain` 的条件熵如下所示：

$$\begin{aligned}H(R\mid M) &= -\frac{1}{12}\big(0.25\log_2(0.25)+0.75\log_2(0.75)\big)-\frac{1}{12}\big(0.27\log_2(0.27)+0.73\log_2(0.73)\big)\\ &\quad -\cdots-\frac{1}{12}\big(0.23\log_2(0.23)+0.77\log_2(0.77)\big)\\ &\approx 0.626\text{ bits}\end{aligned}$$

因此，我们通过月份减少了信号的随机性，但减少得并不多！

给定 `rain` 的情况下，我们还可以计算出 `month` 的条件熵，它表示在知道下雨天的情况下，我们还需要多少信息才能确定月份。直观来看，这样比一无所知的情况更好，因为冬季更可能下雨。

练习：计算条件熵

计算在知道下雨天的情况下月份的条件熵。月份变量的熵是什么（忽略月份的天数差异）？哪个更大？



表中的概率是给定月份时下雨的条件概率。

```
prains = np.array([25, 27, 24, 18, 14, 11, 7, 8, 10, 15, 18, 23]) / 100
pshine = 1 - prains
p_rain_g_month = np.column_stack([prains, pshine])
# 用无条件列联表的表达式替换下面的'None'。提示：表中值的总和必须为1
p_rain_month = None
# 将你计算H(M|R)和H(M)的代码加在下面
```

以下两个值加在一起就定义了信息变异。

$$VI(A,B) = H(A \mid B) + H(B \mid A)$$

5.7 图像分割中的信息论：信息变异

回到图像分割语境，“日期”就变成了“像素”，“下雨”和“月份”就变成了“自动分割标签（S）”和“真实结果标签（T）”。给定真实结果的情况下，如果知道一个像素在 T 中的标识，那么自动分割条件熵用于测量还需要多少信息才能确定像素在 S 中的标识。举例来说，如果每个 T 中的分区 g 能再分为两个大小相等的 S 中的分区 a_1 和 a_2，那么 $H(S|T) = 1$。这是因为，知道一个像素在 g 中之后，你还需要一个额外的位来确定它是属于 a_1 还是 a_2。但是，$H(T|S) = 0$，因为一个像素无论是在 a_1 还是 a_2 中，它都肯定是在 g 中，所以除了知道分区在 S 中，无须更多信息。

因此，在这种情况下：

$$VI(S,T) = H(S \mid T) + H(T \mid S) = 1 + 0 = 1 \text{ bit}$$

以下是一个简单的示例。

```
S = np.array([[0, 1],
              [2, 3]], int)

T = np.array([[0, 1],
              [0, 1]], int)
```

这是对一个 4 像素图像的两种分割方式：S 和 T。S 将每个像素都划为一个分区，T 则将左边两个像素划为分区 0，右边两个像素划为分区 1。接下来，和处理垃圾邮件预测标签一样，我们做一个像素标签的列联表。唯一的区别是，这里的标签数组是二维的，不是一维的预测数组。实际上，这并不重要：记住，NumPy 数组实际上是带有形状和其他元数据的线性数据块。正如前面所说，我们可以用数组的 .ravel() 方法忽略形状。

```
S.ravel()
```

```
array([0, 1, 2, 3])
```

现在我们就可以像预测垃圾邮件时那样做出列联表了。

```
cont = sparse.coo_matrix((np.broadcast_to(1., S.size),
                          (S.ravel(), T.ravel())))
cont = cont.toarray()
cont
```

```
array([[ 1.,  0.],
       [ 0.,  1.],
       [ 1.,  0.],
       [ 0.,  1.]])
```

为了做出概率表，而不是计数表，再除以一下像素总数就可以了。

```
cont /= np.sum(cont)
```

最后，我们可以用这张列联表通过轴向加总计算出标签在 S 或 T 中的概率。

```
p_S = np.sum(cont, axis=1)
p_T = np.sum(cont, axis=0)
```

使用 Python 代码计算熵时有个小问题：尽管 0 log(0) 定义为等于 0，但在 Python 中，它是未定义的，并会得到一个 nan 值（不是一个数值）。

```
print('The log of 0 is: ', np.log2(0))
print('0 times the log of 0 is: ', 0 * np.log2(0))

The log of 0 is:  -inf
0 times the log of 0 is:  nan
```

因此，我们必须用 NumPy 索引功能剔除 0 值。此外，取决于输入是 NumPy 数组还是 SciPy 稀疏矩阵，还需要进行一些不同的处理。为了方便处理，我们使用以下函数。

```
def xlog1x(arr_or_mat):
    """Compute the element-wise entropy function of an array or matrix.

    Parameters
    ----------
    arr_or_mat : numpy array or scipy sparse matrix
    The input array of probabilities. Only sparse matrix formats with a
    `data` attribute are supported.

Returns
-------
out : array or sparse matrix, same type as input
    The resulting array. Zero entries in the input remain as zero,
    all other entries are multiplied by the log (base 2) of their
    inverse.
"""
out = arr_or_mat.copy()
if isinstance(out, sparse.spmatrix):
    arr = out.data
else:
    arr = out
nz = np.nonzero(arr)
arr[nz] *= -np.log2(arr[nz])
return out
```

确认一下代码是否有效。

```
a = np.array([0.25, 0.25, 0, 0.25, 0.25])
xlog1x(a)
```

```
array([ 0.5,  0.5,  0. ,  0.5,  0.5])

mat = sparse.csr_matrix([[0.125, 0.125, 0.25,    0],
                         [0.125, 0.125,    0, 0.25]])
xlog1x(mat).A

array([[ 0.375,  0.375,  0.5  ,  0.   ],
       [ 0.375,  0.375,  0.   ,  0.5  ]])
```

因此，给定 T 时 S 的条件熵如下所示。

```
H_ST = np.sum(np.sum(xlog1x(cont / p_T), axis=0) * p_T)
H_ST

1.0
```

反之：

```
H_TS = np.sum(np.sum(xlog1x(cont / p_S[:, np.newaxis]), axis=1) * p_S)
H_TS

0.0
```

5.8 转换NumPy数组代码以使用稀疏矩阵

前面的示例使用了 NumPy 数组和广播机制，正如我们多次见到的，这是在 Python 中进行数据分析的一种非常强大的方法。然而，对于可能包含几千个分区的复杂图像分割来说，这种方法的效率很快会下降。作为一种替代方案，我们可以用 `sparse` 进行全部计算，并将 NumPy 中的一些奇妙功能改组为线性代数操作。这个建议是在 StackOverflow 网站上由 Warren Weckesser 提供的，参见“Substitute for Numpy Broadcasting Using `scipy.sparse.csc_matrix`”。

线性代数修改过的函数可以有效地为海量数据（最高可达到几十亿个点）计算列联矩阵，而且优雅简洁。

```
import numpy as np
from scipy import sparse


def invert_nonzero(arr):
    arr_inv = arr.copy()
    nz = np.nonzero(arr)
    arr_inv[nz] = 1 / arr[nz]
    return arr_inv


def variation_of_information(x, y):
    # 计算列联矩阵，即联合概率矩阵
    n = x.size
    Pxy = sparse.coo_matrix((np.full(n, 1/n), (x.ravel(), y.ravel())),
                            dtype=float).tocsr()
```

```
    # 计算边际概率，并转换为一维数组
    px = np.ravel(Pxy.sum(axis=1))
    py = np.ravel(Pxy.sum(axis=0))

    # 用稀疏矩阵线性代数计算信息变异
    # 首先，计算逆对角矩阵
    Px_inv = sparse.diags(invert_nonzero(px))
    Py_inv = sparse.diags(invert_nonzero(py))

    # 然后，计算熵
    hygx = px @ xlog1x(Px_inv @ Pxy).sum(axis=1)
    hxgy = xlog1x(Pxy @ Py_inv).sum(axis=0) @ py

    # 返回它们的和
    return float(hygx + hxgy)
```

可以用前面示例中的 `S` 和 `T` 测试一下，看它能否给出信息变异的正确值（1）。

```
variation_of_information(S, T)
```

```
1.0
```

你可以看到，我们使用了 3 种类型的稀疏矩阵（COO、CSR 和对角阵），以便在列联矩阵稀疏的情况下有效地计算熵，此时 NumPy 的效率非常低。（实际上，Python 的 `MemoryError` 异常促使我们实现了这一整套方法！）

5.9 使用信息变异

最后来演示一下信息变异的应用，以估计一幅图像的最优自动分割。你可能还记得第 3 章中那只正在潜行的老虎朋友（见图 5-2）。（如果忘记了，那么你应该提高一下自己的风险评估能力了！）通过使用第 3 章中介绍的技能，我们可以生成分割这幅图像的一些可行方法，然后找出一个最佳方法。

```
from skimage import io

url = ('http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds'
       '/BSDS300/html/images/plain/normal/color/108073.jpg')
tiger = io.imread(url)

plt.imshow(tiger);
```

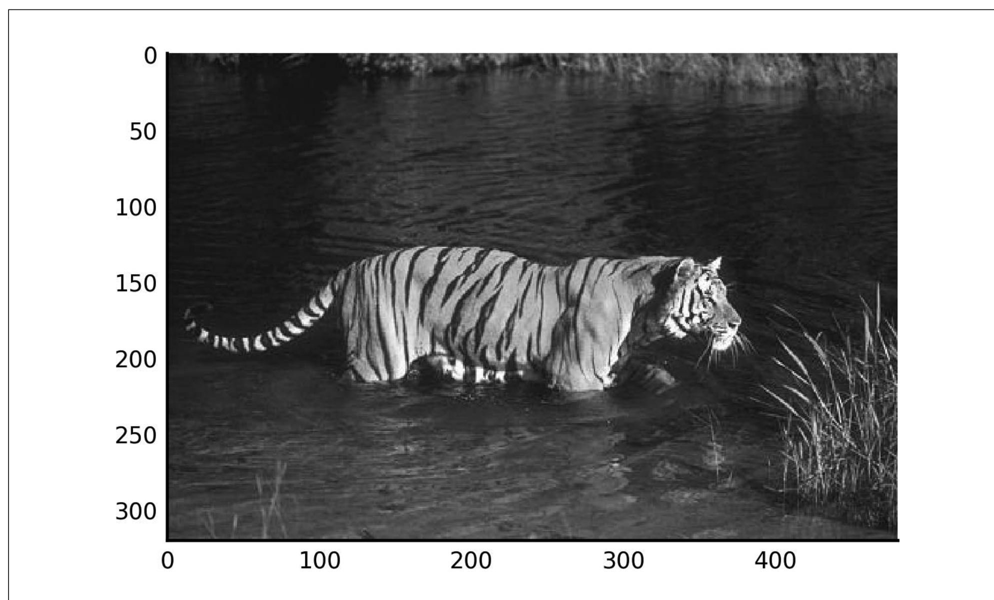


图 5-2：BSDS 老虎图像，编号 108073

为了检验图像分割效果，我们需要某种真实结果。事实证明，人类具有极好的老虎识别能力（使人类最终胜出的自然选择），因此我们需要做的就是让一个人找到老虎。好在伯克利大学的研究者已经请许多人看过这幅图像，并进行了手工分割。[3]

我们看看来自伯克利分割数据集和基准项目（Berkeley Segmentation Dataset and Benchmark）的一种图像分割（见图 5-3）。值得注意的是，人类所做的图像分割之间存在大量变异。如果仔细检查各种不同的老虎分割结果，就会发现有些人比其他人更注重细节，他们将芦苇的轮廓都画出来了；有些人则认为老虎在水中的倒影也有必要和水的其他部分分割。我们选择了一个最喜欢的分割（画出了芦苇轮廓的分割，因为我们是那种追求完美的科学家）。但必须说明的是，真的没有唯一的真实结果！

```
from scipy import ndimage as ndi
from skimage import color

human_seg_url = ('http://www.eecs.berkeley.edu/Research/Projects/CS/'
                 'vision/bsds/BSDS300/html/images/human/normal/'
                 'outline/color/1122/108073.jpg')
boundaries = io.imread(human_seg_url)
plt.imshow(boundaries);
```

注 3：ARBELAEZ P, MAIRE M, FOWLKES C, et al. Contour detection and hierarchical image segmentation [C]// IEEE TPAMI, 2011, 33(5): 898-916.

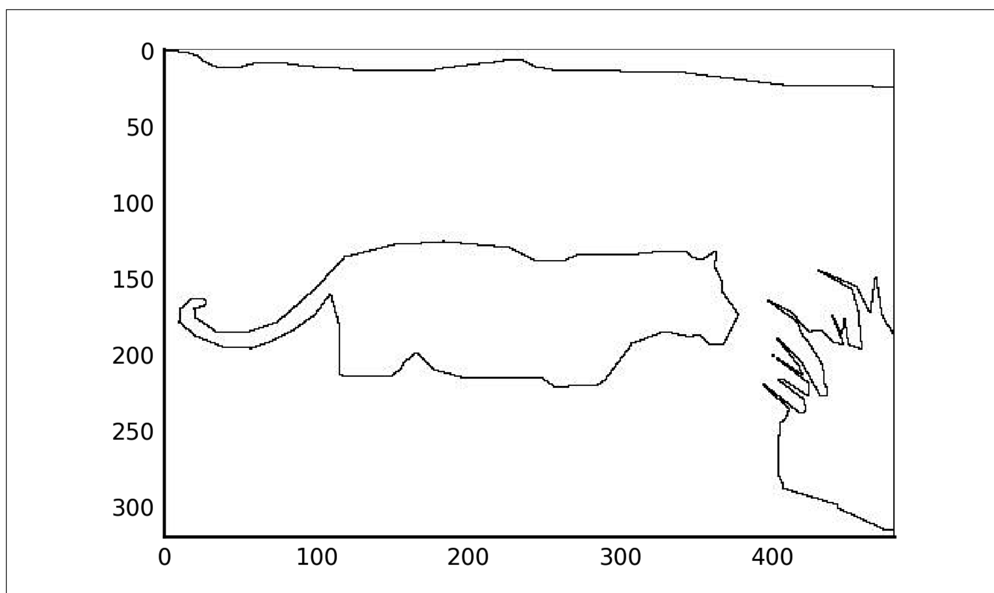


图 5-3：老虎图像的人工分割

将人工分割结果覆盖到老虎图像上，我们可以（不出意料地）看出这个人非常成功地找到了老虎（见图 5-4），他还分割出了河岸和一丛芦苇。干得漂亮，编号为 1122 的人类！

```
human_seg = ndi.label(boundaries > 100)[0]
plt.imshow(color.label2rgb(human_seg, tiger));
```

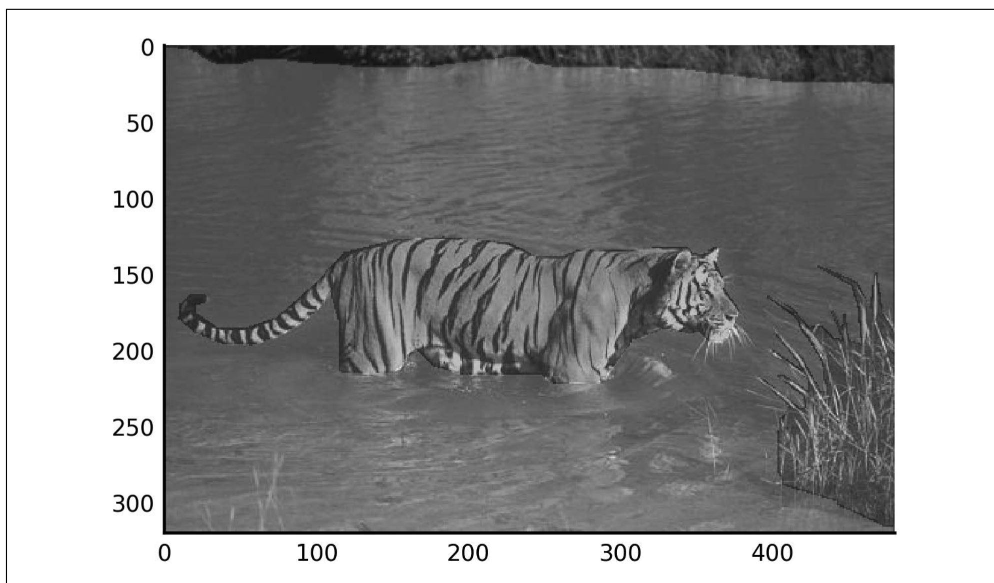


图 5-4：覆盖后的老虎图像的人工分割

现在我们看一下第 3 章中的图像分割代码，看看 Python 能在老虎识别这件事上做得多好（见图 5-5）！

```
# 画出一张RAG——所有代码都来自第3章
import networkx as nx
import numpy as np
from skimage.future import graph

def add_edge_filter(values, graph):
    current = values[0]
    neighbors = values[1:]
    for neighbor in neighbors:
        graph.add_edge(current, neighbor)
    return 0. # generic_filter需要一个返回值，但实际上我们不使用这个值

def build_rag(labels, image):
    g = nx.Graph()
    footprint = ndi.generate_binary_structure(labels.ndim, connectivity=1)
    for j in range(labels.ndim):
        fp = np.swapaxes(footprint, j, 0)
        fp[0, ...] = 0 # 将每个轴最上面的footprint设为0
    _ = ndi.generic_filter(labels, add_edge_filter, footprint=footprint,
                           mode='nearest', extra_arguments=(g,))
    for n in g:
        g.node[n]['total color'] = np.zeros(3, np.double)
        g.node[n]['pixel count'] = 0
    for index in np.ndindex(labels.shape):
        n = labels[index]
        g.node[n]['total color'] += image[index]
        g.node[n]['pixel count'] += 1
    return g

def threshold_graph(g, t):
    to_remove = ((u, v) for (u, v, d) in g.edges(data=True)
                 if d['weight'] > t)
    g.remove_edges_from(to_remove)
# 基准分割
from skimage import segmentation
seg = segmentation.slic(tiger, n_segments=30, compactness=40.0,
                        enforce_connectivity=True, sigma=3)
plt.imshow(color.label2rgb(seg, tiger));
```

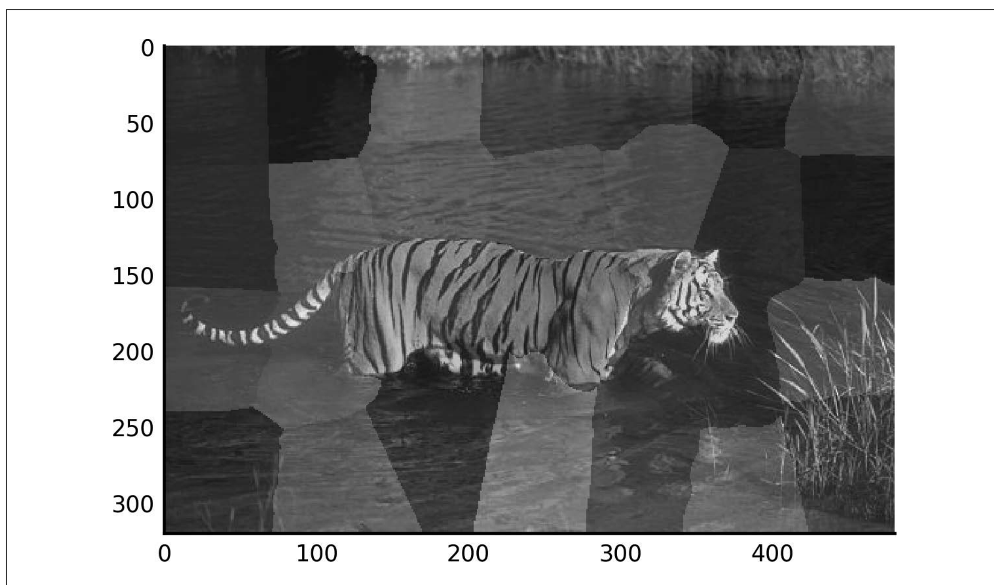


图 5-5：老虎图像的基准 SLIC 分割

第 3 章中设定图的阈值为 80，没有给出任何解释，颇有点蒙混过关的意思。现在我们来仔细研究一下，看看这个阈值对图像分割的准确度到底有什么影响。我们将分割代码写成一个函数，以便使用。

```
def rag_segmentation(base_seg, image, threshold=80):
    g = build_rag(base_seg, image)
    for n in g:
        node = g.node[n]
    node['mean'] = node['total color'] / node['pixel count']
for u, v in g.edges_iter():
    d = g.node[u]['mean'] - g.node[v]['mean']
    g[u][v]['weight'] = np.linalg.norm(d)

threshold_graph(g, threshold)

map_array = np.zeros(np.max(seg) + 1, int)
for i, segment in enumerate(nx.connected_components(g)):
    for initial in segment:
        map_array[int(initial)] = i
segmented = map_array[seg]
return(segmented)
```

我们试验几个阈值，并看看分别是什么情况（见图 5-6 和图 5-7）。

```
auto_seg_10 = rag_segmentation(seg, tiger, threshold=10)
plt.imshow(color.label2rgb(auto_seg_10, tiger));
```

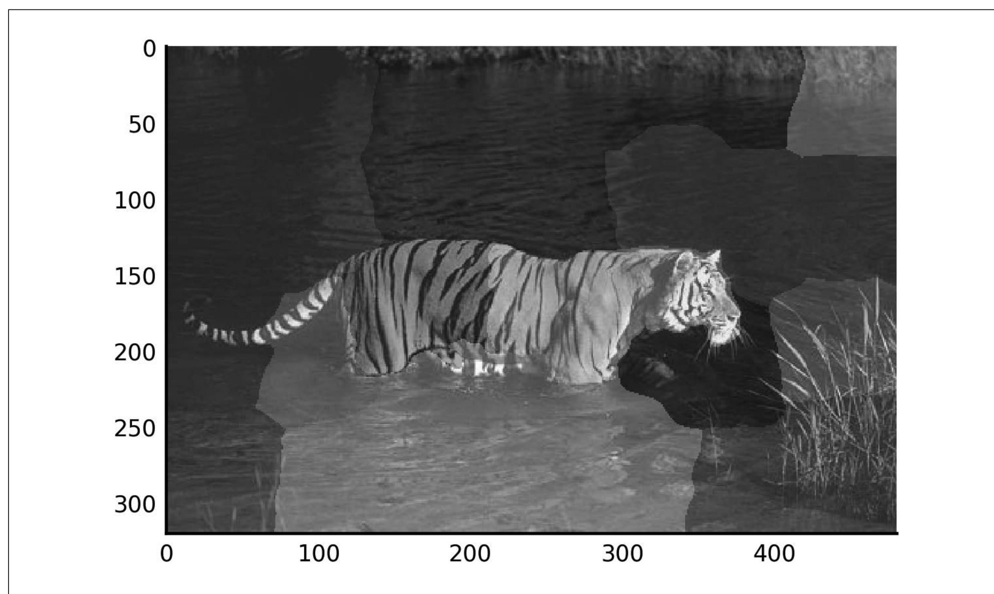


图 5-6：基于 RAG 的老虎图像分割，阈值为 10

```
auto_seg_40 = rag_segmentation(seg, tiger, threshold=40)
plt.imshow(color.label2rgb(auto_seg_40, tiger));
```

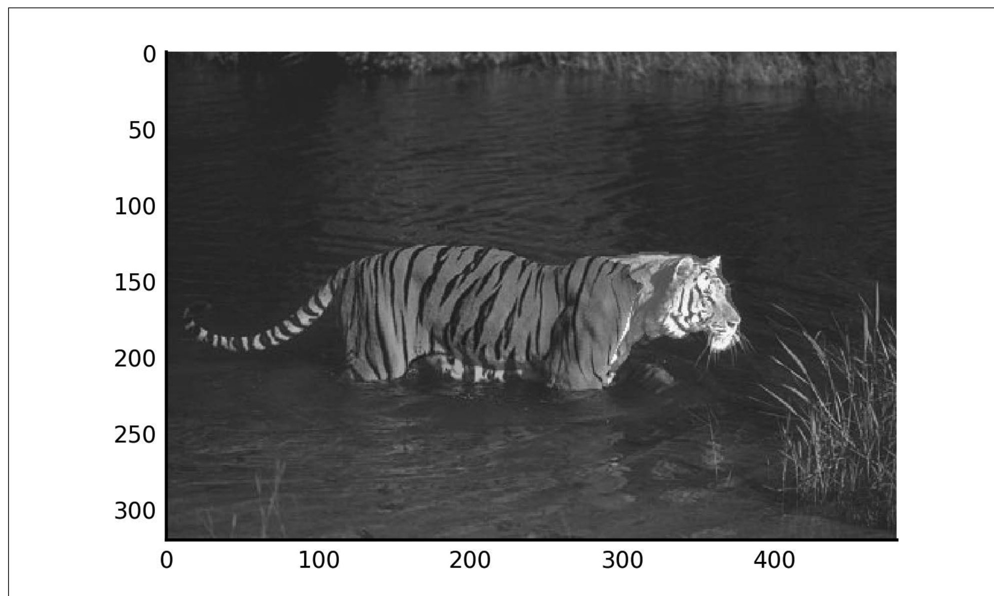


图 5-7：基于 RAG 的老虎图像分割，阈值为 40

实际上，第 3 章中也使用了不同的阈值进行多次分割，然后选择了一个分割效果特别好的（因为我们是人类，所以可以这么做），但对于程序图像分割来说，这种方法完全行不通。显然，我们需要一种自动评价方法。

现在看来，似乎阈值越高，分割效果就越好。但因为有真实结果，所以我们可以算出一个确切的数值！通过使用所有的稀疏矩阵技能，我们可以为每个分割结果计算出信息变异。

```
variation_of_information(auto_seg_10, human_seg)

3.44884607874861

variation_of_information(auto_seg_40, human_seg)

1.0381218706889725
```

高阈值具有较小的信息变异，因此是更好的分割！现在可以为可能范围内的阈值计算信息变异，然后看看哪个阈值能得到与人工真实结果最接近的分割（见图 5-8）。

```
# 试验多个阈值
def vi_at_threshold(seg, tiger, human_seg, threshold):
    auto_seg = rag_segmentation(seg, tiger, threshold)
    return variation_of_information(auto_seg, human_seg)
    thresholds = range(0, 110, 10)
    vi_per_threshold = [vi_at_threshold(seg, tiger, human_seg, threshold)
                        for threshold in thresholds]

    plt.plot(thresholds, vi_per_threshold);
```

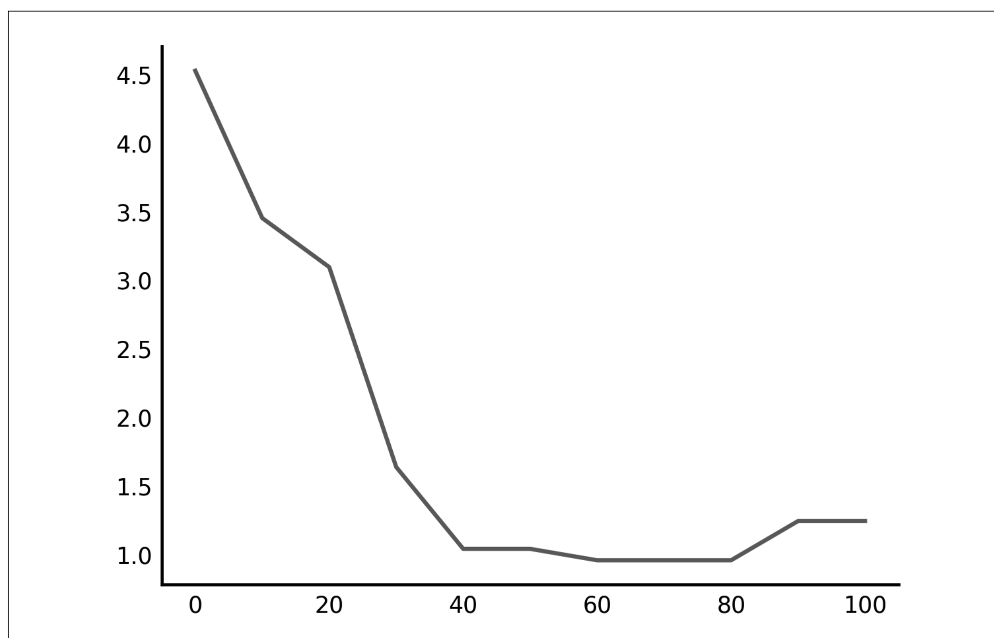


图 5-8：分割结果信息变异与阈值的关系

不出所料，从图中可以看出，选择 threshold = 80 确实可以得出最好的分割结果之一（见图 5-9）。但更重要的是，我们得到了一种可以自动分割任意图像的方法！

```
auto_seg = rag_segmentation(seg, tiger, threshold=80)
plt.imshow(color.label2rgb(auto_seg, tiger));
```

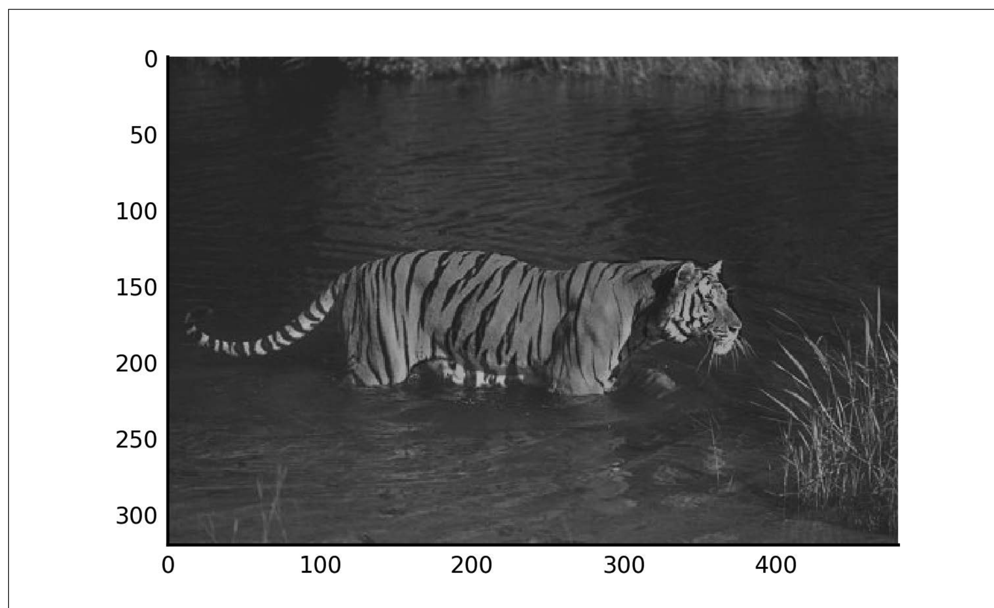


图 5-9：基于信息变异曲线的老虎图像最优分割

进一步工作：图像分割实战

从伯克利分割数据集和基准项目中再选择一些图片，并试着找出它们的最佳阈值。用这些阈值的均值或中位数来分割一幅新图片。你能得到一个合理的分割结果吗？[4]

有时数据中会有很多缺口，这种情形在现实生活中极其常见，稀疏矩阵是表示这种数据的一种有效方式。学习完本章后，你可能已经开始意识到何时可以使用稀疏矩阵，而且也知道了如何使用。

特别适合稀疏矩阵大显身手的一种情形是稀疏线性代数。继续学习下一章，你将发现稀疏矩阵的更多用途！

注 4：ARBELAEZ P, MAIRE M, FOWLKES C, et al. Contour detection and hierarchical image segmentation [C]// IEEE TPAMI, 2011, 33(5): 898-916.

第 6 章

SciPy中的线性代数

> 没有人能告诉你矩阵是什么。你必须亲眼见到它。
>
> ——孟菲斯，《黑客帝国》

第 4 章介绍了快速傅里叶变换，与第 4 章一样，本章将主要介绍一种优雅的**方法**。我们将重点介绍 SciPy 中线性代数运算所用的包，它为很多科学计算功能提供了基础。

6.1 线性代数基础

编程图书实在不是学习线性代数本身的好地方，因此我们假设你已经熟悉线性代数的基本概念。至少你应该知道线性代数处理的主要是向量（数值的有序集合），并通过与矩阵（向量的集合）相乘来对向量进行转换。如果你对这些概念一无所知，那就应该在学习本章前找一本线性代数导论之类的教科书看一下。我们强烈推荐 Gil Strang 的《线性代数及其应用》一书。尽管如此，你了解一下相关基础知识就足够了，我们希望本章内容能够体现出线性代数的强大功能，同时保持操作相对简单！

另外，为了符合线性代数的习惯，我们将打破 Python 的命名惯例。Python 中的变量名通常以小写字母开头。但在线性代数中，矩阵用大写字母表示，向量和标量用小写字母表示。由于我们要处理相当多的矩阵和向量，遵循线性代数的习惯会更加简洁明了。因此，表示矩阵的变量将以大写字母开头，而向量和数值将以小写字母开头，如以下代码所示。

```
import numpy as np
m, n = (5, 6) # 标量
M = np.ones((m, n)) # 矩阵
v = np.random.random((n,)) # 向量
w = M @ v # 另一个向量
```

在数学符号中，向量通常用黑斜体表示，如 $\boldsymbol{v}$ 和 $\boldsymbol{w}$；标量则用白斜体表示，如 m 和 n。Python 代码中无法体现这种区别，因此我们将通过上下文来区分向量和标量。

6.2 图的拉普拉斯矩阵

我们在第 3 章中讨论过图，将图像区域表示为节点，节点之间用边连接。但我们使用的分析方法相当简单，只对图设定**阈值**，然后删除所有高于某个值的边。这种阈值化在简单的情况下很有效，但很容易失败，只要有一个值落在阈值错误的一边，就可能导致方法失败。

举个例子，假设你正处于一场战争中，敌军与你们隔河对峙。为了不让敌人过来，你决定炸掉你们之间所有的桥。根据情报，每炸毁一座桥需要 t 千克 TNT 炸药，但你自己领土上的桥能承受 t+1 千克炸药。学习完第 3 章后，或许你可以命令突击队员在这一区域的每座桥上都引爆 t 千克 TNT 炸药。但如果其中**一座**桥的情报有误，导致它没有被炸毁，那么敌人就会长驱直入了！这简直是一场灾难！

因此，本章将基于线性代数介绍其他几种分析图的方法。研究表明，我们可以将图 G 看作**邻接矩阵**（adjacency matrix），将图中节点从 0 到 $n-1$ 进行编号，只要节点 i 和节点 j 之间有一条边，那么就在矩阵的第 i 行第 j 列放一个 1。换句话说，如果有一个邻接矩阵 $\boldsymbol{A}$，那么 $\boldsymbol{A}_{i,j}=1$，当且仅当 G 中有一条边 (i, j)。然后我们可以用线性代数技术来研究矩阵，这样经常能得到非常显著的成果。

节点的**度**定义为与其相连的边的数量。例如，在一个图中，如果一个节点与其他 5 个节点相连，那么它的度就是 5。（稍后我们还要根据边是从节点“出发”还是“到达”来区分“出度”和“入度”。）用矩阵术语来说，度对应一行或一列中值的**总和**。

图的**拉普拉斯**矩阵（简称拉普拉斯矩阵）的定义是**度矩阵** $\boldsymbol{D}$（对角线上是每个节点的度，其余元素为 0）与邻接矩阵 $\boldsymbol{A}$ 的差。

$$\boldsymbol{L}=\boldsymbol{D}-\boldsymbol{A}$$

我们肯定不能用所有相关线性代数理论来解释这个矩阵的性质，但知道它确实有一些非常好的性质就够了。我们将在后面的内容中使用其中的几个性质。

首先看一下 $\boldsymbol{L}$ 的**特征向量**（eigenvector）。矩阵 $\boldsymbol{M}$ 的特征向量 $\boldsymbol{v}$ 是一个能满足以下条件的向量：对于某个值 λ，$\boldsymbol{Mv}=\lambda\boldsymbol{v}$，$\lambda$ 称为特征值。换句话说，$\boldsymbol{v}$ 是一个与 $\boldsymbol{M}$ 相关的特殊向量，因为 $\boldsymbol{Mv}$ 只改变了向量的大小，而没有改变其方向。正如我们即将看到的，特征向量具有很多有用的性质，有时甚至非常奇妙！

我们来看一个示例。当一个 3×3 的旋转矩阵 $\boldsymbol{R}$ 乘以任意一个三维向量 $\boldsymbol{p}$ 时，可以将 $\boldsymbol{p}$ 绕 z 轴旋转 30 度。除了位于 z 轴上的向量，$\boldsymbol{R}$ 可以旋转所有向量。z 轴上的向量看不到任何效果，用线性代数来表示就是 $\boldsymbol{Rp}=\boldsymbol{p}$（即 $\boldsymbol{Rp}=\lambda\boldsymbol{p}$），特征值 $\lambda=1$。

练习：旋转矩阵

思考下面的旋转矩阵。

$$
\boldsymbol{R}=\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ \sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}
$$

当 $\boldsymbol{R}$ 乘以一个三维列向量 $\boldsymbol{p}=[x\ y\ z]^{\mathrm{T}}$ 时，结果向量 $\boldsymbol{Rp}$ 会绕 z 轴旋转 θ 度。

(1) 令 θ 为 45 度，通过试验任意几个向量，验证 $\boldsymbol{R}$ 使向量绕 z 轴旋转的效果。记住，Python 用 `@` 表示矩阵乘法。

(2) 矩阵 $\boldsymbol{S}=\boldsymbol{RR}$ 的作用是什么？用 Python 验证一下。

(3) 验证向量 $[0\ 0\ 1]^{\mathrm{T}}$ 乘以 $\boldsymbol{R}$ 后保持不变；换句话说，$\boldsymbol{Rp}=1\boldsymbol{p}$，这说明 $\boldsymbol{p}$ 是 $\boldsymbol{R}$ 特征值为 1 的特征向量。

(4) 用 `np.linalg.eig` 找出 $\boldsymbol{R}$ 的特征值和特征向量，并验证 $[0\ 0\ 1]^{\mathrm{T}}$ 确实是一个特征向量，而且对应的特征值为 1。

回到拉普拉斯矩阵。可视化是网络分析中的一个常见问题。节点和边应该如何绘制，才能不像图 6-1 中那样一团乱麻呢？

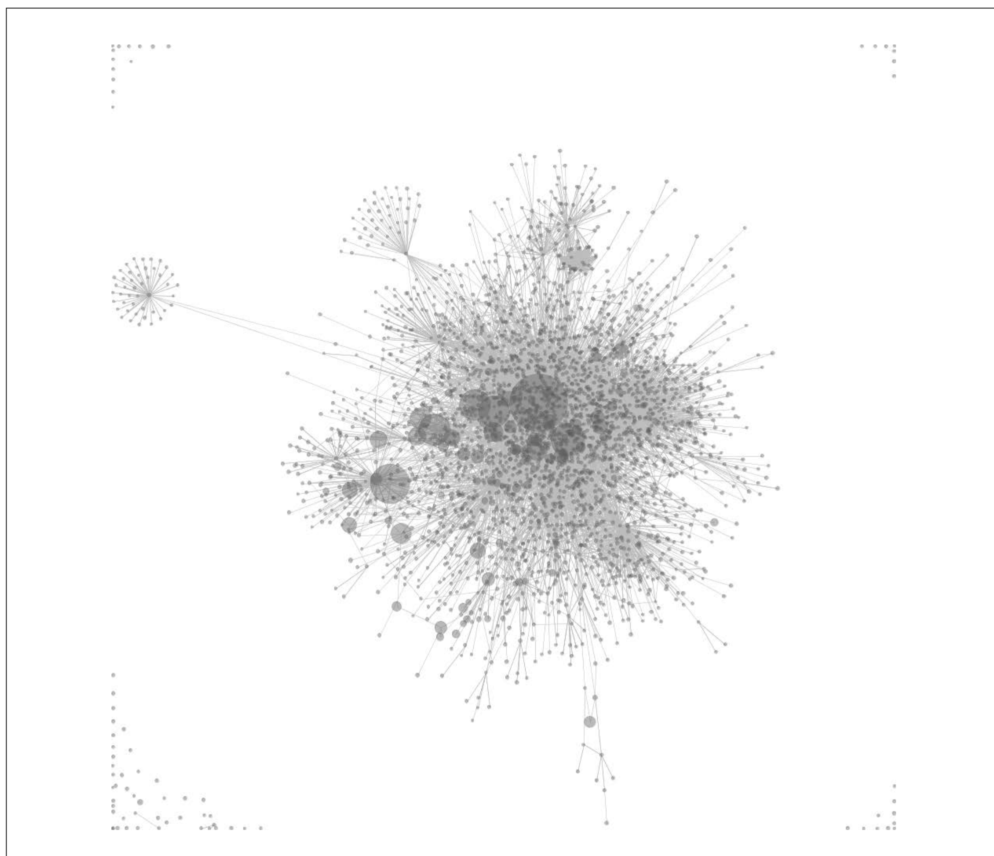


图 6-1：维基百科结构的可视化（由 Chris Davis 制作，遵循 CC-BY-SA-3.0 许可协议发布）

一种方法是将共享多个边的节点放在一起。事实证明，要想完成这个任务，我们可以使用拉普拉斯矩阵第二小的特征值及其对应的特征向量，这个特征向量太重要了，以至于有一个专有名称：Fiedler 向量。

接下来用一个最小的网络来演示一下，从创建邻接矩阵开始。

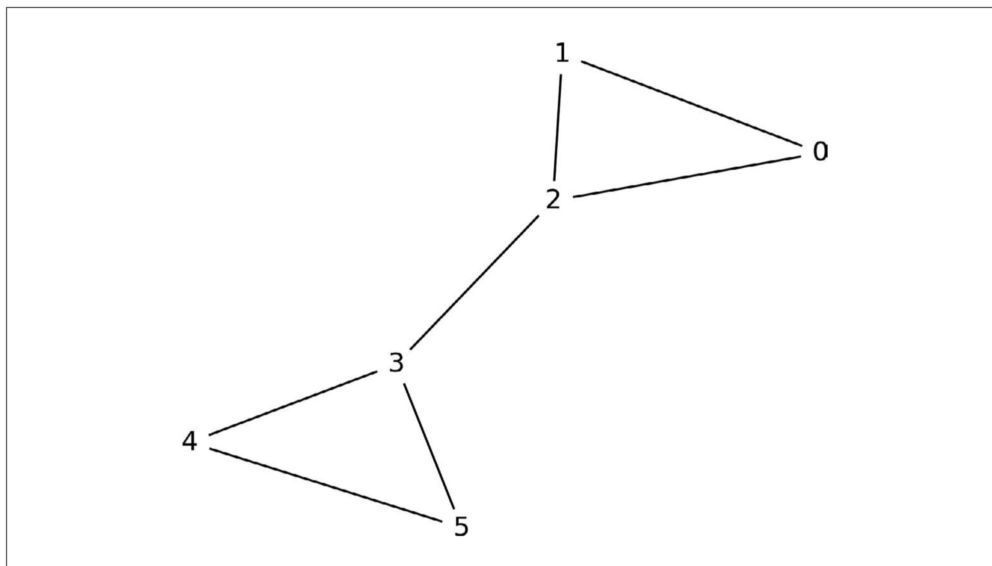
```
import numpy as np
A = np.array([[0, 1, 1, 0, 0, 0],
              [1, 0, 1, 0, 0, 0],
              [1, 1, 0, 1, 0, 0],
              [0, 0, 1, 0, 1, 1],
              [0, 0, 0, 1, 0, 1],
              [0, 0, 0, 1, 1, 0]], dtype=float)
```

可以用 NetworkX 画出这个网络。首先，像往常一样初始化 Matplotlib。

```
# 使图形显示在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

现在就可以绘制出网络图了。

```
import networkx as nx
g = nx.from_numpy_matrix(A)
layout = nx.spring_layout(g, pos=nx.circular_layout(g))
nx.draw(g, pos=layout,
        with_labels=True, node_color='white')
```



从上图可以看到，结点自动分成了两个组：0、1、2 和 3、4、5。Fiedler 向量能告诉我们这种情况吗？第一步，我们必须计算出度矩阵和拉普拉斯矩阵。通过沿着 $\boldsymbol{A}$ 的任意一个轴求和，我们先得到度。（任意一个轴都可以，因为 $\boldsymbol{A}$ 是对称的。）

```
d = np.sum(A, axis=0)
print(d)
[ 2.  2.  3.  3.  2.  2.]
```

然后将这些度放在与 A 形状相同的一个对角阵中，这就是**度矩阵**。可以用 `np.diag` 函数来完成这个任务。

```
D = np.diag(d)
print(D)

[[ 2.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.]
 [ 0.  0.  3.  0.  0.  0.]
 [ 0.  0.  0.  3.  0.  0.]
 [ 0.  0.  0.  0.  2.  0.]
 [ 0.  0.  0.  0.  0.  2.]]
```

最后根据定义得出拉普拉斯矩阵。

```
L = D - A
print(L)

[[ 2. -1. -1.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.]
 [-1. -1.  3. -1.  0.  0.]
 [ 0.  0. -1.  3. -1. -1.]
 [ 0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0. -1. -1.  2.]]
```

因为 L 是对称的，所以可以用 `np.linalg.eigh` 函数计算出特征值和特征向量。

```
val, Vec = np.linalg.eigh(L)
```

我们可以验证返回值，看看它们是否满足特征值和特征向量的定义。例如，其中一个特征值为 3。

```
np.any(np.isclose(val, 3))

True
```

我们还可以验证用矩阵 L 乘以对应的特征向量确实与用 3 乘以该向量相同。

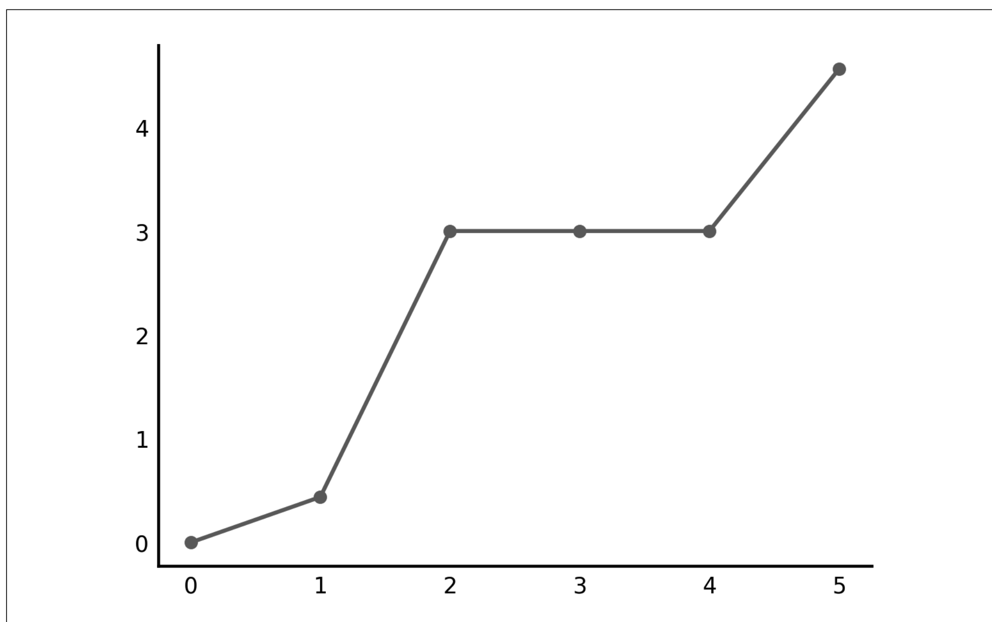
```
idx_lambda3 = np.argmin(np.abs(val - 3))
v3 = Vec[:, idx_lambda3]

print(v3)
print(L @ v3)

[ 0.          0.37796447 -0.37796447 -0.37796447 0.68898224 -0.31101776]
[ 0.          1.13389342 -1.13389342 -1.13389342 2.06694671 -0.93305329]
```

正如前面所说，Fiedler 向量是与 L 的次小特征值对应的特征向量。对特征值排序可以找出次小的特征值。

```
plt.plot(np.sort(val), linestyle='-', marker='o');
```



这是第一个非零特征值，接近 0.4。Fiedler 向量就是与这个特征值相对应的特征向量（见图 6-2）。

```
f = Vec[:, np.argsort(val)[1]]
plt.plot(f, linestyle='-', marker='o');
```

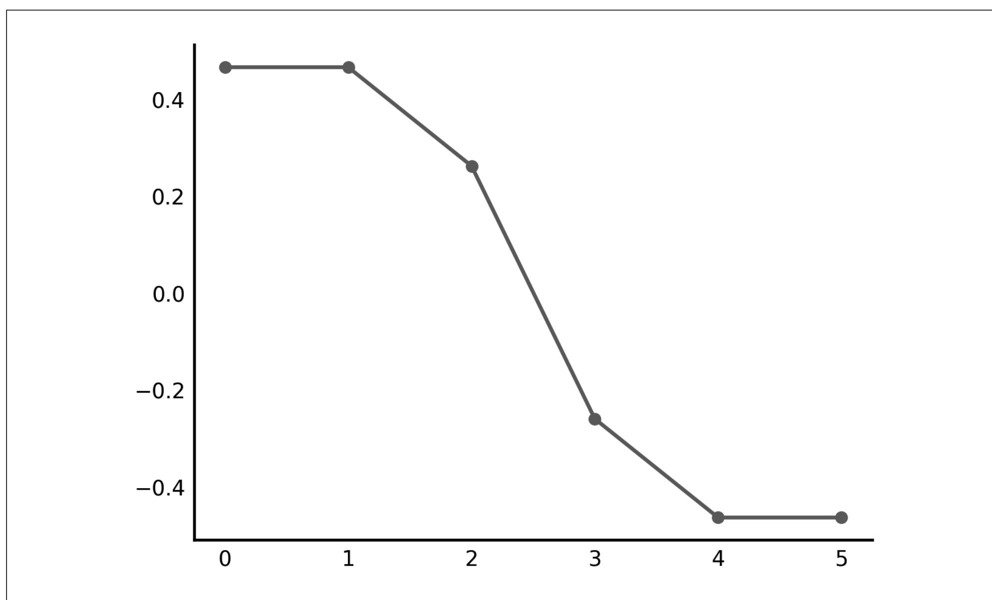


图 6-2：*L* 的 Fiedler 向量

非常明显：通过 Fiedler 向量中各元素的**符号**，我们可以将节点分成图 6-3 中的两个组！

```
colors = ['orange' if eigv > 0 else 'gray' for eigv in f]
nx.draw(g, pos=layout, with_labels=True, node_color=colors)
```

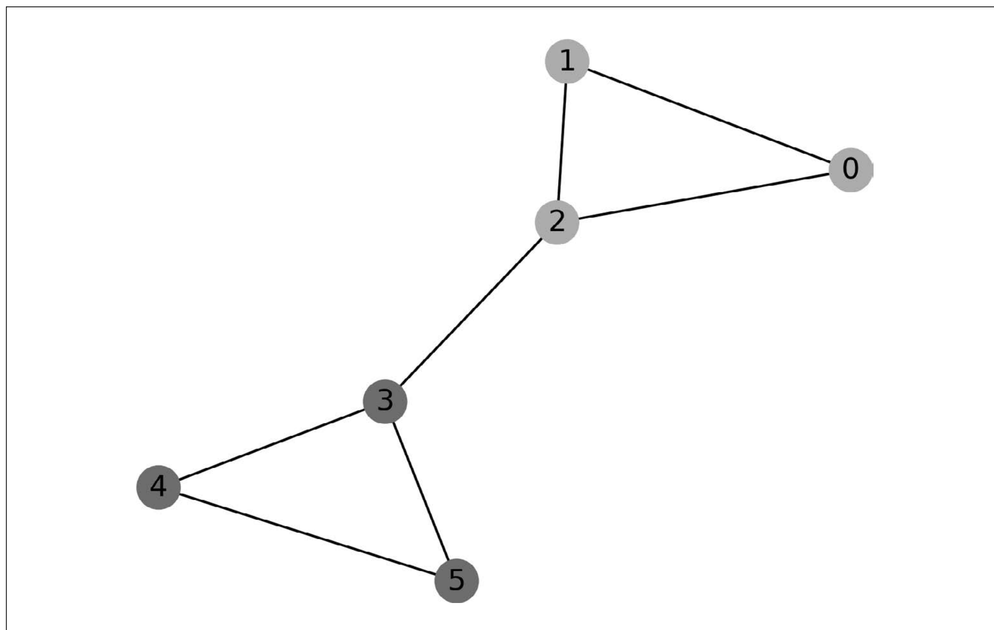


图 6-3：*L* 的 Fiedler 向量中按符号着色的节点

6.3　大脑数据的拉普拉斯矩阵

我们通过一个实际示例来演示这个过程，生成一张蠕虫脑细胞分布图，也就是第 3 章中提到的 Varshney 等人的论文中的图 2（关于如何完成这张图的信息，参见论文的补充资料：http://journals.plos.org/ploscompbiol/article/file?id=info:doi/10.1371/journal.pcbi.1001066.s001&type=supplementary）。为了得到蠕虫大脑神经元的分布，他们使用了称为**度标准化拉普拉斯矩阵**的相关矩阵。

因为神经元的顺序在这个分析中非常重要，所以我们使用一个经过预处理的数据集，以避免本章充斥着数据清洗的内容。我们从 Lav Varshney 的个人网站获取了原始数据，并将处理后的数据放在 data/ 目录中。

先加载数据。数据分为如下 4 个部分。

- 化学突触网络，**突触前神经元**可以通过这个网络向**突触后神经元**发射化学信号。
- 间隙连接网络，其中包括神经元之间的电接触。
- 神经元 ID（名称）。
- 3 种神经元类型：
  - **感觉神经元**（sensory neuron）是用于感知来自外部世界的信号的神经元，编码为 0；

- **运动神经元**（motor neuron）是用于刺激肌肉，使蠕虫运动的神经元，编码为 2；
- **中间神经元**（interneuron）是感觉神经元和运动神经元之间的神经元，可以在二者之间处理复杂信号，编码为 1。

```
import numpy as np
Chem = np.load('data/chem-network.npy')
Gap = np.load('data/gap-network.npy')
neuron_ids = np.load('data/neurons.npy')
neuron_types = np.load('data/neuron-types.npy')
```

接下来简化网络，将两种连接合二为一，并通过取神经元入连接和出连接的平均数除去网络的有向性。别担心这样做会有问题，因为我们只是想看看神经元在图上的**布局**，只需要关注神经元之间**是否**有连接，无须在意连接的方向。我们称结果矩阵为**连接矩阵**，用 $\boldsymbol{C}$ 表示，这只是另一种邻接矩阵。

```
A = Chem + Gap
C = (A + A.T) / 2
```

得到拉普拉斯矩阵 $\boldsymbol{L}$，需要度矩阵 $\boldsymbol{D}$，它包含了节点 i 在 $[i, j]$ 位置的度，其他位置均为 0。

```
degrees = np.sum(C, axis=0)
D = np.diag(degrees)
```

和前面一样，现在就可以得到拉普拉斯矩阵。

```
L = D - C
```

排列节点，使神经元在总体上尽可能靠近其下游邻居节点的正上方，这样就可以确定论文图 2 中的纵坐标。Varshney 等人称这种测量方式为“处理深度”，通过解一个由拉普拉斯矩阵构成的线性方程，可以得出处理深度。我们可以使用 `scipy.linalg.pinv`（即伪逆矩阵）来解这个方程。

```
from scipy import linalg
b = np.sum(C * np.sign(A - A.T), axis=1)
z = linalg.pinv(L) @ b
```

（注意 `@` 符号的用法，它是在 Python 3.5 中引入的，用于表示矩阵乘法。正如我们在前言和第 5 章中提到的，更早的 Python 版本中需要使用函数 `np.dot`。）

为了得到度标准化拉普拉斯矩阵 $\boldsymbol{Q}$，我们需要矩阵 $\boldsymbol{D}$ 的平方根的倒数。

```
Dinv2 = np.diag(1 / np.sqrt(degrees))
Q = Dinv2 @ L @ Dinv2
```

最后可以提取出神经元的 x 坐标，以确保高连接神经元集中在一起：用度进行标准化的、与矩阵 $\boldsymbol{Q}$ 的次小特征值对应的特征向量。

```
val, Vec = linalg.eig(Q)
```

注意 `numpy.linalg.eig` 文档中的以下提示：

> 特征值不一定是排过序的。

尽管 SciPy 的 `eig` 文档中没有这条警告，但确实是这样的。因此，我们自己必须对特征值及其对应的特征向量进行排序。

```
smallest_first = np.argsort(val)
val = val[smallest_first]
Vec = Vec[:, smallest_first]
```

这样就可以找出计算近邻坐标所需的特征向量。

```
x = Dinv2 @ Vec[:, 1]
```

（使用这个向量的原因说来话长，具体内容请参见之前的论文补充资料链接。简而言之，选择这个向量可以使神经元之间的链接总长度最短。）

在进行下一步之前，必须先说明一个小问题：特征向量就是一个可乘的常量。这可以根据特征向量的定义推导出来。假设 $\boldsymbol{v}$ 是矩阵 $\boldsymbol{M}$ 的一个特征向量，对应的特征值是 λ，那么对于任意标量值 α，$\alpha\boldsymbol{v}$ 也是 $\boldsymbol{M}$ 的一个特征向量，因为 $\boldsymbol{M}\boldsymbol{v} = \lambda\boldsymbol{v}$ 意味着 $\boldsymbol{M}(\alpha\boldsymbol{v}) = \lambda(\alpha\boldsymbol{v})$。因此，用软件包计算 $\boldsymbol{M}$ 的特征向量时，它既可能返回 $\boldsymbol{v}$，也可能返回 $-\boldsymbol{v}$。为了确保能重新生成 Varshney 等人论文中的布局，我们必须确定所有向量都指向与原文相同的方向，而不是相反的方向。要想达到这个目的，我们可以任意选择图 2 中的一个神经元，并检查该位置上 `x` 的符号，如果与图 2 中的符号不符，则取其相反数。

```
vc2_index = np.argwhere(neuron_ids == 'VC02')
if x[vc2_index] < 0:
    x = -x
```

现在就只剩下节点和边的绘制了。我们按照 `neuron_types` 中保存的类型为节点着色，使用既好看又实用的 colorbrewer 调色板。

```
from matplotlib.colors import ListedColormap
from matplotlib.collections import LineCollection


def plot_connectome(x_coords, y_coords, conn_matrix, *,
                    labels=(), types=None, type_names=('',),
                    xlabel='', ylabel=''):
    """Plot neurons as points connected by lines.

    Neurons can have different types (up to 6 distinct colors).

    Parameters
    ----------
    x_coords, y_coords : array of float, shape (N,)
        The x-coordinates and y-coordinates of the neurons.
    conn_matrix : array or sparse matrix of float, shape (N, N)
        The connectivity matrix, with nonzero entry (i, j) if and only
        if node i and node j are connected.
    labels : array-like of string, shape (N,), optional
        The names of the nodes.
    types : array of int, shape (N,), optional
        The type (e.g. sensory neuron, interneuron) of each node.
    type_names : array-like of string, optional
        The name of each value of `types`. For example, if a 0 in
```

```
        `types` means "sensory neuron", then `type_names[0]` should
        be "sensory neuron".
    xlabel, ylabel : str, optional
        Labels for the axes.
    """
    if types is None:
        types = np.zeros(x_coords.shape, dtype=int)
    ntypes = len(np.unique(types))
    colors = plt.rcParams['axes.prop_cycle'][:ntypes].by_key()['color']
    cmap = ListedColormap(colors)

    fig, ax = plt.subplots()

    # 绘制神经元位置
    for neuron_type in range(ntypes):
        plotting = (types == neuron_type)
        pts = ax.scatter(x_coords[plotting], y_coords[plotting],
                         c=cmap(neuron_type), s=4, zorder=1)
        pts.set_label(type_names[neuron_type])

    # 添加文本标签
    for x, y, label in zip(x_coords, y_coords, labels):
        ax.text(x, y, '   ' + label,
                verticalalignment='center', fontsize=3, zorder=2)

    # 绘制边
    pre, post = np.nonzero(conn_matrix)
    links = np.array([[x_coords[pre], x_coords[post]],
                      [y_coords[pre], y_coords[post]]]).T
    ax.add_collection(LineCollection(links, color='lightgray',
                                     lw=0.3, alpha=0.5, zorder=0))

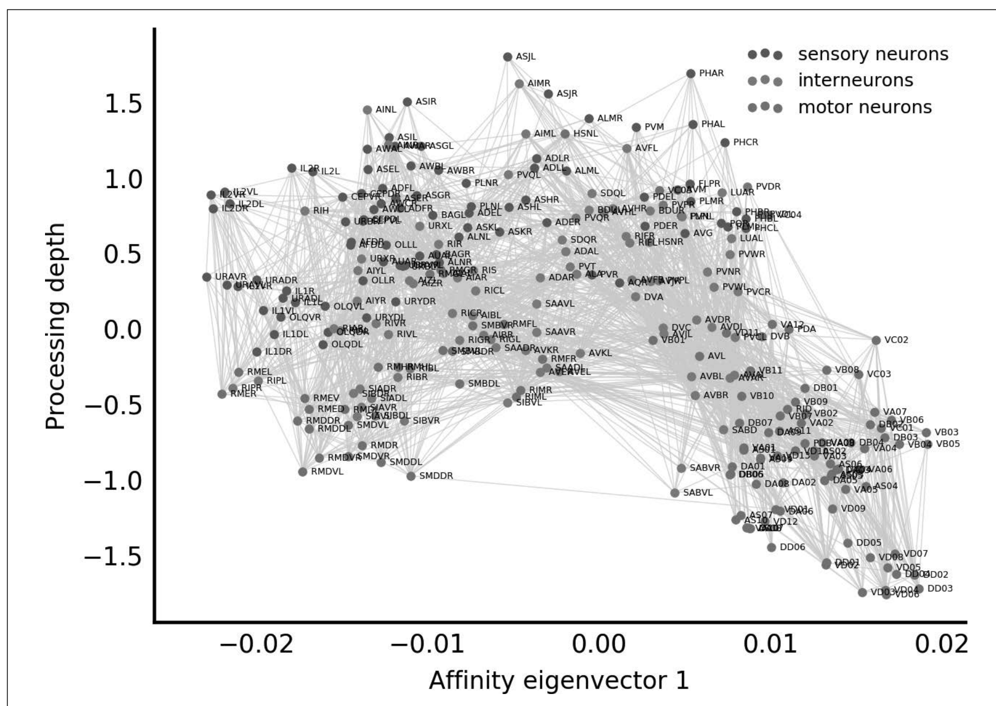
    ax.legend(scatterpoints=3, fontsize=6)

    ax.set_xlabel(xlabel, fontsize=8)
    ax.set_ylabel(ylabel, fontsize=8)

    plt.show()
```

现在用这个函数来绘制神经元，如以下代码和图片所示。

```
plot_connectome(x, z, C, labels=neuron_ids, types=neuron_types,
                type_names=['sensory neurons', 'interneurons',
                            'motor neurons'],
                xlabel='Affinity eigenvector 1', ylabel='Processing depth')
```



就是这样，一个蠕虫的大脑！正如原文所述，你可以看到从感觉神经元开始，经由中间神经元网络，再到运动神经元的这种自顶向下的处理过程。你还可以看到运动神经元中有两个明显的分组，分别对应蠕虫的脖颈（左侧）和身体（右侧）。

6.3.1　练习：显示近邻视图

如何修改以上代码来显示论文图 2B 中的近邻视图呢？

6.3.2　练习挑战：稀疏矩阵线性代数

以上代码用 NumPy 数组来保存矩阵并执行必要的计算。因为我们使用的是一个比较小的图，不到 300 个节点，所以这样做是可行的。但对于更大的图，这种方法会失效。

例如，有人可能想分析 PyPI（即 Python 包索引，其中包括 10 000 多个 Python 包）中列出的各个库之间的联系，那么保存这个图的拉普拉斯矩阵就需要 $8 \times (100 \times 10^3)^2 = 8 \times 10^{10}$ 字节，即 80 GB 的内存。如果再加上邻接矩阵、对称邻接矩阵、伪逆矩阵等计算中需要的临时矩阵，所需内存就会超过 480 GB，这已经超过了绝大多数台式计算机的内存极限。

可能有人会这样想：“哼，我的电脑有 512 GB 的内存！对付这种所谓的‘大’图还不是小菜一碟！”

可能是这样的。但如果你想分析美国计算机协会的引用图呢？这个网络包含超过 200 万条学术文献和引用条目，其拉普拉斯矩阵大概需要 32 TB 的内存。

然而我们知道，表示这种依赖和引用的图是**稀疏的**：一个包通常只依赖于其他几个包，论文与专著通常也只引用其他几种论文和专著。因此，可以用 `scipy.sparse`（见第 5 章）中的稀疏数据结构来保存上面所说的矩阵，并用 `scipy.sparse.linalg` 中的线性代数函数来计算需要的值。

研究一下 `scipy.sparse.linalg` 的文档，实现上面计算过程的一个稀疏矩阵版本。



稀疏矩阵的伪逆一般不是稀疏的，因此这里不能使用伪逆矩阵。同理，你也不能得到一个稀疏矩阵的所有特征向量，因为它们会一起构成一个稠密矩阵。

你可以在下面的附注栏中找到部分解决方案（答案当然在附录中），但我强烈建议你自己尝试一下。

求解程序

SciPy 中有若干种支持稀疏矩阵的可迭代求解程序，有时并不确定应该使用哪一种。遗憾的是，这个问题也很难回答，因为不同的算法在收敛速度、稳定性、准确性和内存使用等方面各有千秋。即使检查了输入数据，也不太可能预测出哪种算法的效果最好。

以下是选择可迭代求解程序的大体原则。

- 如果输入矩阵 A 是对称并正定的，那么可以使用共轭梯度求解程序 `cg`。如果 A 是对称的，但是近奇异或不定矩阵，那么可以尝试最小残差迭代方法 `minres`。
- 对于非对称系统，可以试一下双共轭梯度稳定方法 `bicgstab`。共轭梯度平方法 `cgs` 速度会快一点，但收敛速度不太稳定。
- 如果需要求解很多相似系统，可以使用 LGMRES 算法 `lgmres`。
- 如果 A 不是方阵，可以使用最小二乘算法 `lsmr`。

延伸阅读参见：

- Noël M. Nachtigal、Satish C. Reddy 和 Lloyd N. Trefethen 的文章“How Fast are Nonsymmetric Matrix Iterations?”，1992 年发表于 *SIAM Journal on Matrix Analysis and Applications*，第 13 卷，第 3 期，第 778~795 页；
- Jack Dongarra 的文章“Survey of Recent Krylov Methods”，发表于 1995 年 11 月 20 日。

6.4 PageRank：用于声望和重要性的线性代数

线性代数与特征向量的另一个应用是 Google 的 PageRank 算法，这个算法的名称既表示跟网页有关，又包含了 Google 联合创始人 Larry Page 的名字，非常有趣。

为了按照重要性对网页排名，你可以计算一下与之相连的其他网页的数量。毕竟，如果所有网页都链接到某个网页，那这个网页肯定很好，不是吗？但这种度量方式很容易被钻空

子。想要使自己的网页排名上升，那就创建尽可能多的网页，并让它们都链接到初始网页就可以了。

使 Google 获得初步成功的核心洞见是，重要的网页不是那些被很多网页链接的网页，而是被**重要网页**链接的网页。那么，怎么知道其他网页是否重要呢？看它们本身是否也被重要网页链接。以此类推。

这个递归定义意味着网页的重要性可以通过所谓的**转移矩阵**（transition matrix）的特征向量来测量，这个转移矩阵中包含了网页之间的链接。假设你有一个表示网页重要性的向量 $\boldsymbol{r}$，还有一个表示网页链接的矩阵 $\boldsymbol{M}$。你还不知道 $\boldsymbol{r}$ 的具体值，但知道一个网页的重要性与那些链接到它的网页的重要性之和成正比：$\boldsymbol{r} = \alpha \boldsymbol{M}\boldsymbol{r}$，或 $\boldsymbol{M}\boldsymbol{r} = \lambda \boldsymbol{r}$，$\lambda = 1/\alpha$。这恰好是特征值的定义！

通过确认转移矩阵满足一些特殊性质，我们可以进一步确定所需的特征值是 1，这是 $\boldsymbol{M}$ 的最大特征值。

转移矩阵假设有一个网页浏览者，通常称为 Webster，他随机地点击所访问网页上的一个链接，然后找出他访问到给定网页的概率，这个概率就称为 PageRank。

由于 Google 的兴起，研究者将 PageRank 算法应用到了各种网络。接下来使用的示例来自 Stefano Allesina 和 Mercedes Pascual 的文章“Googling Food Webs: Can an Eigenvector Measure Species’Importance for Coextinctions?”，发表在 *PLoS Computational Biology* 期刊上。他们用这种方法来研究生态**食物网**，即将物种及其食物链接起来的网络。

简单地说，如果你想知道一个物种在某个生态系统中的重要性，那么就应该看看有多少物种以它为食物。如果以它为食的物种很多，一旦这个物种灭绝了，那么所有“依赖”这个物种的物种就会随之灭绝。用网络的术语来说，物种的**入度**决定其生态重要性。

PageRank 是生态系统重要性的一种更好的度量方式吗？

Allesina 教授贴心地提供了几个食物网络供我们试验。我们用图形标记语言格式保存了其中一个，这个网络表示的是位于佛罗里达州的圣马可国家野生动物保护区的生态系统。这个网络是由 Robert R. Christian 和 Joseph J. Luczovich 于 1999 年建立的，参见他们的文章“Organizing and understanding a winter’s seagrass foodweb network through effective trophic levels”。在这个数据集中，如果物种 i 以物种 j 为食，那么节点 i 就有一条指向节点 j 的边。

下面先加载数据，用 NetworkX 来读取：

```
import networkx as nx

stmarks = nx.read_gml('data/stmarks.gml')
```

然后建立与图对应的稀疏矩阵。因为矩阵只保存数值型数据，所以还要维护一个与矩阵的行/列对应的名称列表。

```
species = np.array(stmarks.nodes()) # 用于多重索引的数组
Adj = nx.to_scipy_sparse_matrix(stmarks, dtype=np.float64)
```

根据这个邻接矩阵，可以导出**转移概率**矩阵，其中每条边都替换为一个**概率**，即 1 除以从这个物种发出的边的数量。在食物网中，或许更应该称其为午餐概率矩阵。

因为矩阵中的物种总数会多次用到，所以用变量 n 来表示它。

```
n = len(species)
```

下一步需要出度矩阵，具体说是一个**对角矩阵**，其对角线上是每个节点出度的倒数。

```
np.seterr(divide='ignore') # 忽略除零错误
from scipy import sparse

degrees = np.ravel(Adj.sum(axis=1))
Deginv = sparse.diags(1 / degrees).tocsr()

Trans = (Deginv @ Adj).T
```

一般情况下，PageRank 分数就是转移矩阵的第一个特征向量。如果转移矩阵为 $\boldsymbol{M}$，PageRank 值向量为 $\boldsymbol{r}$，则有：

$$\boldsymbol{r} = \boldsymbol{M}\boldsymbol{r}$$

但根据 `np.seterr` 函数来看，事情没这么简单。PageRank 算法仅在转移矩阵为**列随机矩阵**时才有效，即矩阵每列的和为 1。此外，每个网页对于其他网页来说都应该是可达的，即使到达路径特别长也是如此。

然而在食物网中，这会引起一些问题，这是因为被论文作者称为**碎屑**（detritus，实际上是海洋淤泥）的食物链底层实际上不**吃**任何东西（尽管如此，它还是参与生命循环的），因此无法从这里到达其他物种。

> 小辛巴："可是，爸爸，难道我们不吃羚羊吗？"
>
> 木法沙："我们吃，辛巴，但让我解释一下。我们死了以后，尸体就会变成青草；羚羊就会来吃青草。我们就是这样互相连接，共同存在于这个巨大的生命轮回之中。"
>
> ——《狮子王》

PageRank 用"阻尼系数"来处理这种情况，通常它的值是 0.85。这意味着，在 85% 的时间内，算法会随机地沿着链接浏览网页，但在其余 15% 的时间内，它会随机地跳到任意网页。似乎每个网页都有一个到所有其他网页的低概率链接。以食物网为例，就是虾在极其罕见的情况下会吃鲨鱼。这似乎不合常理，但请相信我们，这真的就是生命轮回的数学表示。我们将阻尼系数设为 0.85，但实际上这个具体数值对我们的分析不太重要：阻尼系数的较大取值范围，分析结果都很接近。

如果阻尼系数为 d，那么修正后的 PageRank 公式是：

$$r = d\boldsymbol{M}\boldsymbol{r} + \frac{1-d}{n}\mathbf{1}$$

并且：

$$(\boldsymbol{I} - d\boldsymbol{M})\,\boldsymbol{r} = \frac{1-d}{n}\boldsymbol{1}$$

可以用 scipy.sparse.linalg 中的 spsolve 直接解这个方程。然而，根据一个线性代数问题的结构和规模，使用可迭代求解程序会更有效率。可以查阅 sicpy.sparse.linalg 的文档来获取关于这个问题的更多信息。

```
from scipy.sparse.linalg import spsolve

damping = 0.85
beta = 1 - damping

I = sparse.eye(n, format='csc') # 和Trans相同的稀疏格式

pagerank = spsolve(I - damping * Trans,
                   np.full(n, beta / n))
```

现在我们就得到了圣马可食物网中的“食物排名”！

那么某个物种的食物排名和以它为食的其他物种数量是什么关系呢？

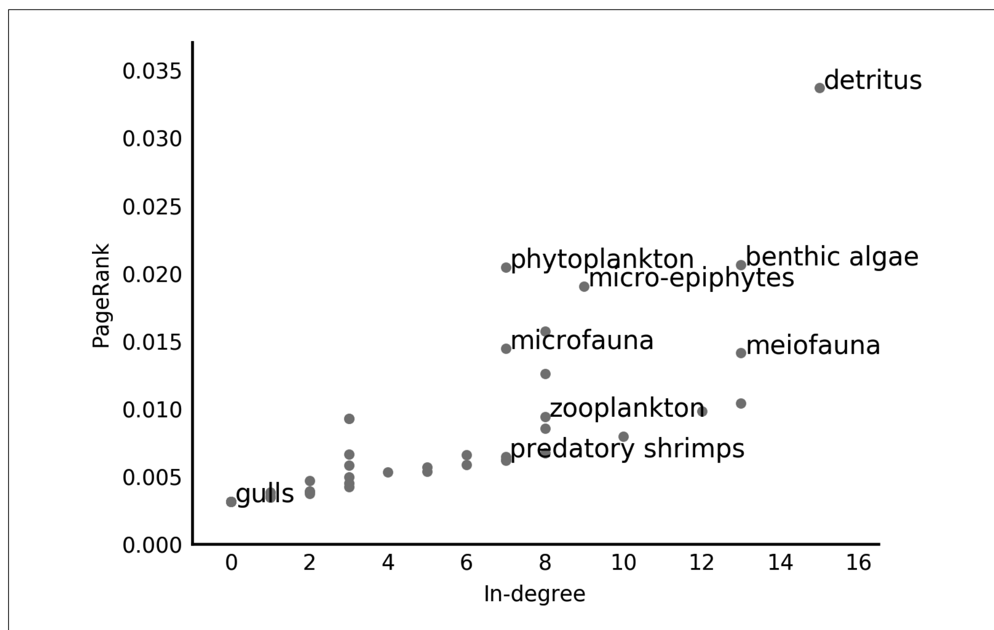
```
def pagerank_plot(in_degrees, pageranks, names, *,
                  annotations=[], **figkwargs):
    """Plot node pagerank against in-degree, with hand-picked node names."""

    fig, ax = plt.subplots(**figkwargs)
    ax.scatter(in_degrees, pageranks, c=[0.835, 0.369, 0], lw=0)
    for name, indeg, pr in zip(names, in_degrees, pageranks):
        if name in annotations:
            text = ax.text(indeg + 0.1, pr, name)

    ax.set_ylim(0, np.max(pageranks) * 1.1)
    ax.set_xlim(-1, np.max(in_degrees) * 1.1)
    ax.set_ylabel('PageRank')
    ax.set_xlabel('In-degree')
```

接下来画出图形。在写这段内容前，我们对数据集进行了一些探索，对图形中一些有趣的节点做了预标记（见下图）。

```
interesting = ['detritus', 'phytoplankton', 'benthic algae', 'micro-epiphytes',
               'microfauna', 'zooplankton', 'predatory shrimps', 'meiofauna',
               'gulls']
in_degrees = np.ravel(Adj.sum(axis=0))
pagerank_plot(in_degrees, pagerank, species, annotations=interesting)
```



从上图来看，海洋淤泥（碎屑，detritus）是生态系统中最重要的元素，不论是从以它为食的物种数量（15）来看，还是从 PageRank（>0.003）来看。但第二重要的元素**不是**供养了 13 个物种的海底藻类（benthic algae），而是只供养了 7 个物种的浮游植物（phytoplankton）！这是因为其他**重要的**物种以浮游植物为食。我们在图的左下角看到了海鸥（gulls），现在可以确认它对生态系统无足轻重。那些丑恶的肉食性虾类（predatory shrimps，我们可不是在胡编乱造）和浮游植物支持的物种数目相同，但它们不是必不可少的物种，因此最终的食物排名非常低。

Allesina 和 Pascual 进一步用模型研究了物种灭绝对生态系统的影响，这里就不再深入讨论了。他们发现，在预测生态系统重要性方面，PageRank 的效果要比入度更好。

结束本章前，我们要说明一下，可以使用几种不同的方法来计算 PageRank。称为**幂法**（power method）的一种方法可以与上面介绍的方法互补，它名副其实，很强大！这种方法源于 Perron-Frobenius 定理，该定理的一部分证明了 1 是随机矩阵的一个特征值，而且是最大的特征值。（对应的特征向量就是 PageRank 向量。）其中的含义是，每当用 $\boldsymbol{M}$ 乘以任意一个向量，它指向主特征向量的成分是不变的，而**其他成分会被一个可乘因子缩减**。结果就是，如果用 $\boldsymbol{M}$ 反复地乘以一个随机初始向量，最终会得到 PageRank 向量！

SciPy 可以通过其稀疏矩阵模块使这个过程非常有效。

```
def power(Trans, damping=0.85, max_iter=10**5):
    n = Trans.shape[0]
    r0 = np.full(n, 1/n)
    r = r0
    for _iter_num in range(max_iter):
```

```
        rnext = damping * Trans @ r + (1 - damping) / n
        if np.allclose(rnext, r):
            break
        r = rnext
    return r
```

6.4.1　练习：处理悬挂节点

注意，以上迭代中的 `Trans` 不是列随机的，因此向量 r 在每次迭代中都会缩减。为了使矩阵是随机的，我们必须把每个 0 列替换为都是 $1/n$ 的列。这样做更容易计算迭代，但代价过大。如何修改以上代码以确保 r 总是概率向量呢？

6.4.2　练习：不同特征向量方法的等价性

验证一下，这 3 种方法会给出同样的节点排名。`numpy.corrcoef` 将是一个有用的函数。

6.5　结束语

线性代数是一个极其宽广的领域，区区一章的篇幅难以介绍详尽，但本章可以让你一窥其强大威力，并了解如何通过 Python、NumPy 和 SciPy 使用线性代数写出优雅的算法。

第7章

SciPy中的函数优化

> “有什么新鲜事儿？”这是一个人们最感兴趣的问题，但是也最不着边际，可以没完没了地问下去。如果认真探讨它的答案，所得的只不过是一堆琐碎的跟风事物，这些都是将来的淤泥。我宁可问这样的问题：“什么是最好的？”这个问题能加深河道而非拓宽，这个问题的答案可以将淤泥冲刷到河流下游。
>
> ——罗伯特·M. 波西格，《禅与摩托车维修艺术》

往墙上挂一幅画，有时很难挂正。你调整了一下，退后检查画是否水平，然后重复这些操作。这就是**优化**的过程：不断改变画的角度，直到满足需求，即与地平线成 0 度角。

用数学语言表示的话，我们的需求称为“损失函数”，画与地平线之间的角度称为“参数”。在典型的优化问题中，我们不断修改参数，直到损失函数最小化。

举例来说，$f(x) = (x - 3)^2$ 是平移了的抛物线，我们要找出使这个损失函数最小化的 x 值。我们知道，这个带有参数 x 的函数在 x 为 3 时取最小值，因为可以先求出它的导数，再设导数为 0，就可以得到 $2(x - 3) = 0$（即 $x = 3$）。

但是，如果这个函数更加复杂（比如，函数表达式有很多项、有多重零导数点、包含非线性，或者依赖多个变量），那么人工计算就会非常困难。

你可以认为损失函数表示一片土地，而我们要找出其最低点。这种类比立刻指出了问题的难点：如果身处某个山谷中，四周都是高山，那么如何才能知道我们是否处于最低的山谷中，或者说，是不是因为这个山谷被高山环绕，所以只是看上去很低？用优化领域的语言来说，就是如何才能知道是否陷入了**局部最小值**？多数优化方法都试图解决这个问题。[1]

注 1：优化算法用各种各样的方法解决这个问题，最常用的两种方法是**线搜索**（line search）和**信赖域**（trust region）。使用线搜索方法时，先设法沿着一个特定维度找到损失函数的最小值，继而在其他维度上做出同样的努力。使用信赖域方法时，先朝着一个预期能得到最小值的方向移动，如果真如预期那样接近了最小值，那么就提高置信度，重复前面的过程；如果没有，则降低置信度，搜索一个更广泛的区域。



图 7-1 给出了 SciPy 中现有的全部方法，我们将使用其中一些方法，其余方法留给读者去发现、学习。

可以选择的优化方法有很多种（见图 7-1）。你可以选择损失函数的输入是标量还是向量（即需要优化的是一个参数还是多个参数）。有些方法需要给定损失函数的梯度，有些方法会自动估计梯度。有些方法只在给定的区域内搜索参数（**约束优化**），有些方法则搜索整个参数空间。

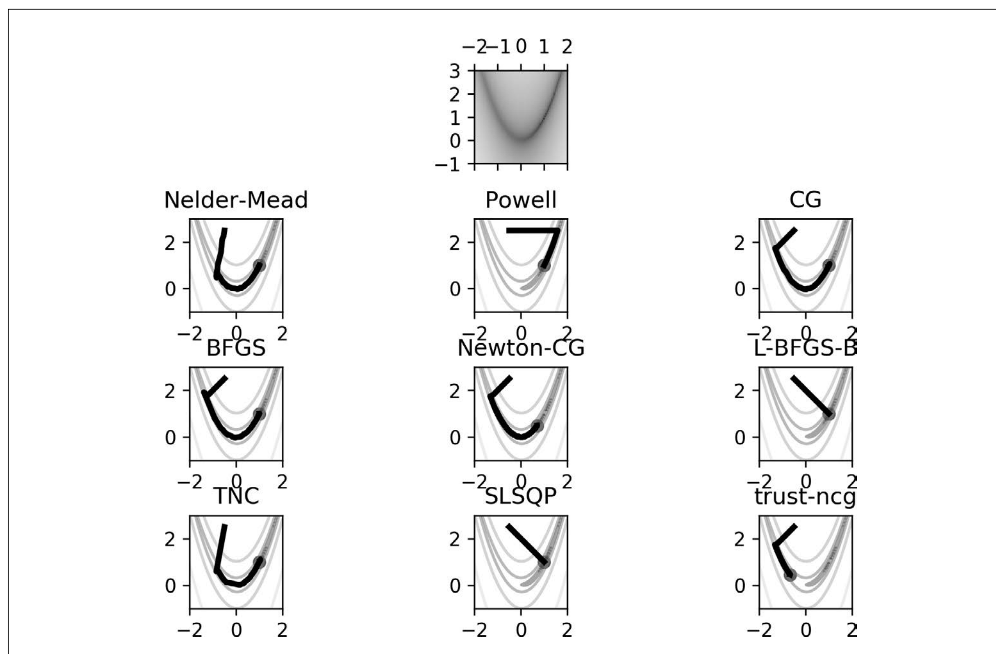


图 7-1：在 Rosenbrock 函数（顶图）上使用不同优化算法的优化路径比较。Powell 方法在梯度下降前沿着第一个维度进行线搜索。另一方面，共轭梯度法从起点执行梯度下降

7.1　SciPy优化模块：sicpy.optimize

本章余下内容将用 SciPy 的 `optimize` 模块来对齐两张图像。图像对齐（或称**图像配准**）的应用包括全景拼接、组合脑扫描、超分辨率成像，在天文学应用中，还有通过组合多次曝光结果进行的目标降噪技术。

和往常一样，先设置绘图环境。

```
# 使图形显示在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')
```

先从最简单的问题开始：有两张图片，其中一张相对于另一张有一定的平移，我们希望通过平移找回最好的图像对齐效果。

我们的优化函数会“抖动”其中一张图片，看看沿着一个方向或另一个方向抖动图片是否会减少二者的差异。不断重复这个操作就能设法找到正确的对齐。

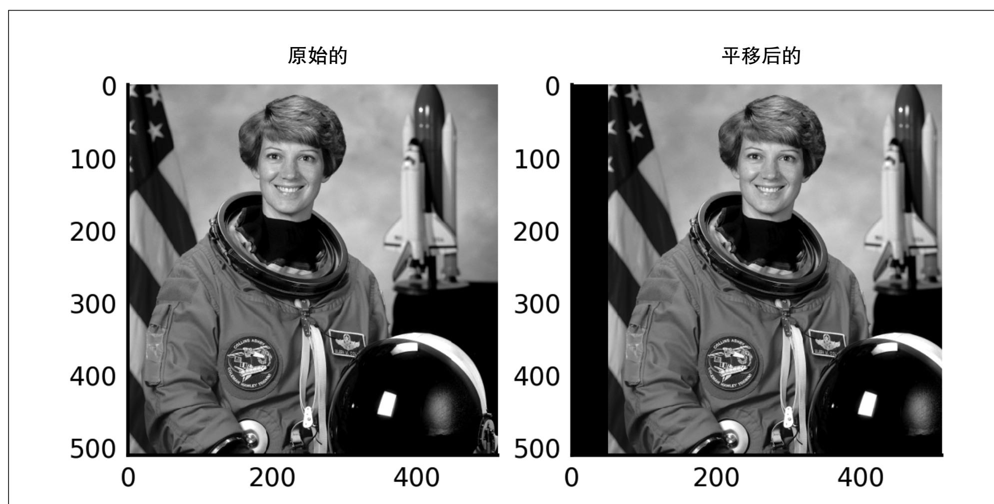
示例：计算最优图片平移

你应该还记得第 3 章中提过的宇航员艾琳 • 柯林斯，我们先将这张图片向右平移 50 个像素，然后再与原始图片比较，直到找出与原始图片匹配得最好的平移。显然，这么做有点傻，因为我们知道原来的位置，但这样就可以知道真实结果，以检查算法的效果。以下是原始图片和平移后的图片。

```
from skimage import data, color
from scipy import ndimage as ndi

astronaut = color.rgb2gray(data.astronaut())
shifted = ndi.shift(astronaut, (0, 50))

fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(astronaut)
axes[0].set_title('Original')
axes[1].imshow(shifted)
axes[1].set_title('Shifted');
```



要想使用优化算法来完成这项任务，需要定义“相异度”，即损失函数。定义相异度的最简方法是计算二者间差的平方的均值，通常称为**均方误差**（MSE，mean squared error）。

```
import numpy as np

def mse(arr1, arr2):
    """Compute the mean squared error between two arrays."""
    return np.mean((arr1 - arr2)**2)
```

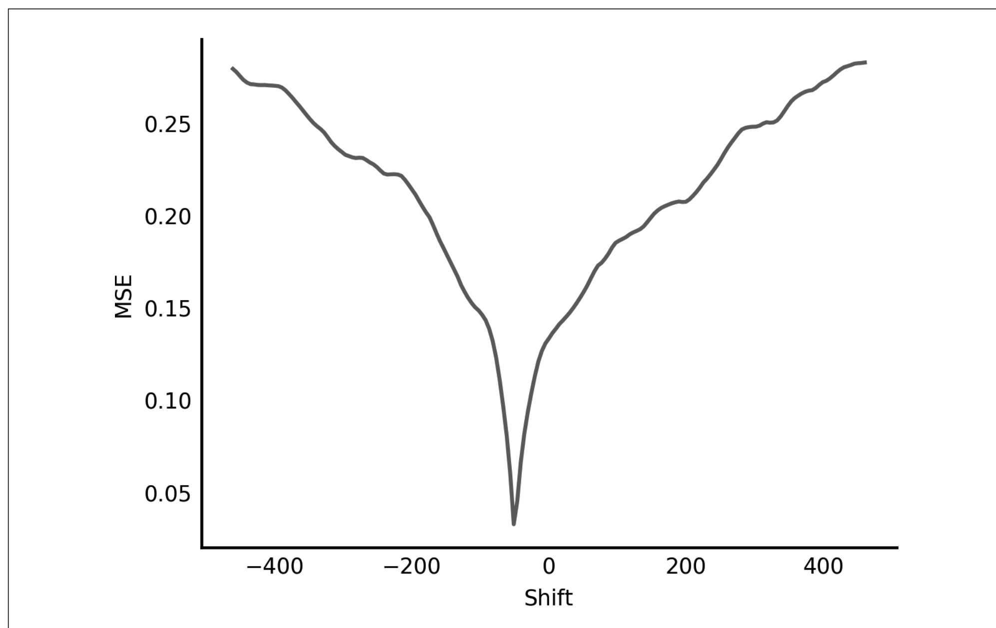
当图片完美对齐时，这个函数会返回 0，否则会返回一个大于 0 的值。可以通过这个损失函数来检查两张图片是否对齐。

```
ncol = astronaut.shape[1]

# 生成一个数列，前后长度都是列长度的90%，每个百分点为一个值
shifts = np.linspace(-0.9 * ncol, 0.9 * ncol, 181)
mse_costs = []

for shift in shifts:
    shifted_back = ndi.shift(shifted, (0, shift))
    mse_costs.append(mse(astronaut, shifted_back))

fig, ax = plt.subplots()
ax.plot(shifts, mse_costs)
ax.set_xlabel('Shift')
ax.set_ylabel('MSE');
```



定义损失函数后，可以用 `scipy.optimize.minimize` 函数来搜索最优参数。

```
from scipy import optimize

def astronaut_shift_error(shift, image):
    corrected = ndi.shift(image, (0, shift))
    return mse(astronaut, corrected)

res = optimize.minimize(astronaut_shift_error, 0, args=(shifted,),
                        method='Powell')

print(f'The optimal shift for correction is: {res.x}')
```

```
The optimal shift for correction is: -49.99997565757551
```

效果不错！我们将图片平移了 50 个像素，通过 MSE 这个指标，SciPy 的 `optimize.minimize` 函数为我们找到了正确的平移量（−50），以便将图片移回初始状态。

事实证明，虽然这是一个非常容易的优化问题，但可能遇到图像对齐中的一种主要难题：有时 MSE 成事不足，败事有余。

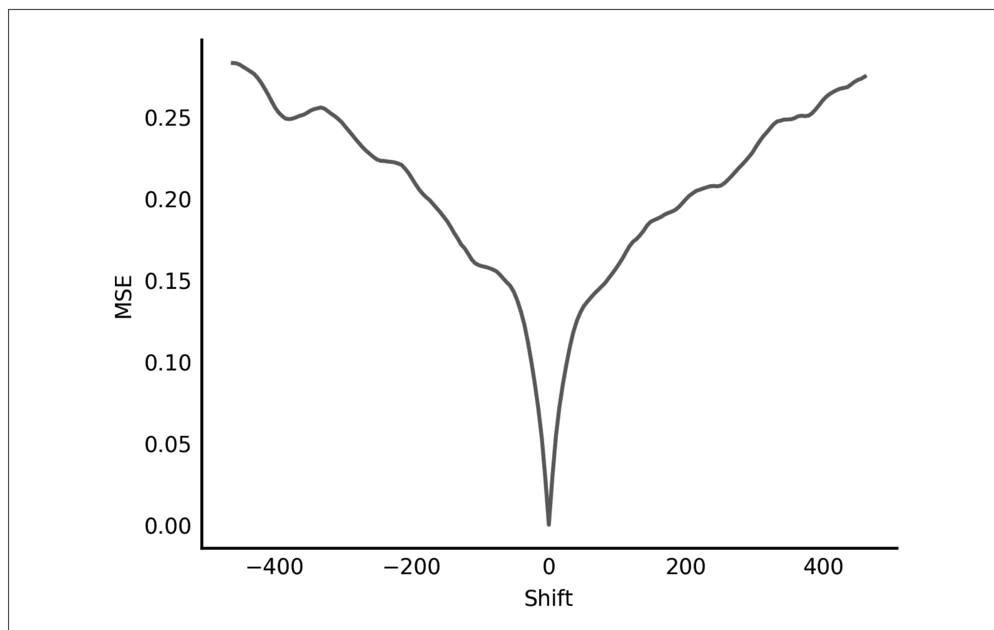
我们再来看一个图像平移问题，这次从未经修改的图像开始。

```
ncol = astronaut.shape[1]

# 生成一个数列，前后长度都是列长度的90%，每个百分点为一个值
shifts = np.linspace(-0.9 * ncol, 0.9 * ncol, 181)
mse_costs = []

for shift in shifts:
    shifted1 = ndi.shift(astronaut, (0, shift))
    mse_costs.append(mse(astronaut, shifted1))

fig, ax = plt.subplots()
ax.plot(shifts, mse_costs)
ax.set_xlabel('Shift')
ax.set_ylabel('MSE');
```



从 0 平移开始，看看逐渐向负方向平移时 MSE 的变化：它一直增加，直至平移到 −300 像素附近才开始减少；虽然很轻微，但毕竟在减少。MSE 在 −400 像素附近降至一个低点，然后又开始上升。这就是**局部最小值**。因为优化方法只能获取损失函数“附近的”值，所以如果向某个方向移动可以改善损失函数时，即使这个方向是“错误的”，`minimize` 过程

还是会不顾一切地向这个方向移动。因此，如果从平移了 −340 像素的图像开始的话：

```
shifted2 = ndi.shift(astronaut, (0, -340))
```

`minimize` 函数会继续将图像平移约 40 像素，而不是回到原始图像。

```
res = optimize.minimize(astronaut_shift_error, 0, args=(shifted2,),
                        method='Powell')

print(f'The optimal shift for correction is {res.x}')

The optimal shift for correction is -38.51778619397471
```

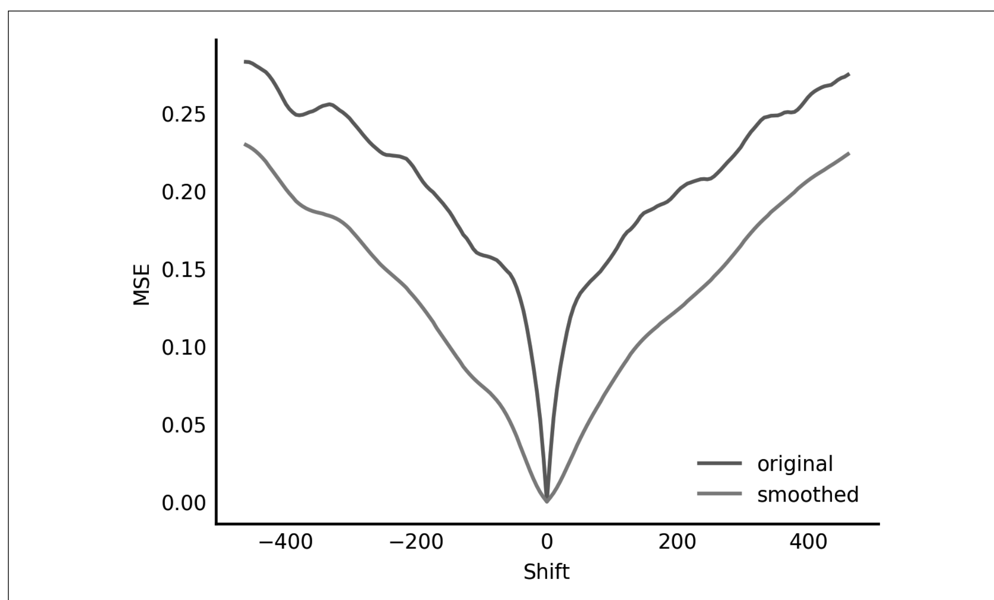
这个问题的一般解决方案是对图像进行平滑和向下缩放，这些操作对目标函数也有平滑效果。对图像使用高斯滤波器进行平滑后，再用同样的方法绘制出图形。

```
from skimage import filters

astronaut_smooth = filters.gaussian(astronaut, sigma=20)

mse_costs_smooth = []
shifts = np.linspace(-0.9 * ncol, 0.9 * ncol, 181)
for shift in shifts:
    shifted3 = ndi.shift(astronaut_smooth, (0, shift))
    mse_costs_smooth.append(mse(astronaut_smooth, shifted3))

fig, ax = plt.subplots()
ax.plot(shifts, mse_costs, label='original')
ax.plot(shifts, mse_costs_smooth, label='smoothed')
ax.legend(loc='lower right')
ax.set_xlabel('Shift')
ax.set_ylabel('MSE');
```



如你所见，经过程度很高的平滑后，误差函数中的“漏斗”变得更宽、更平滑了。除了对函数本身进行平滑，在比较图像前对其进行模糊处理也可以达到同样的效果。因此，现代的图像对齐软件使用一种称为**高斯金字塔**的方法，这种方法可以得到一组分辨率逐渐降低的图像版本。我们先对齐低分辨率（更模糊的）图像，得到一种近似的对齐，然后再用更清晰的图像逐渐改善对齐结果。

```
def downsample2x(image):
    offsets = [((s + 1) % 2) / 2 for s in image.shape]
    slices = [slice(offset, end, 2)
              for offset, end in zip(offsets, image.shape)]
    coords = np.mgrid[slices]
    return ndi.map_coordinates(image, coords, order=1)

def gaussian_pyramid(image, levels=6):
    """Make a Gaussian image pyramid.

    Parameters
    ----------
    image : array of float
        The input image.
    max_layer : int, optional
        The number of levels in the pyramid.

    Returns
    -------
    pyramid : iterator of array of float
        An iterator of Gaussian pyramid levels, starting with the top
        (lowest resolution) level.
    """
    pyramid = [image]

    for level in range(levels - 1):
        blurred = ndi.gaussian_filter(image, sigma=2/3)
        image = downsample2x(image)
        pyramid.append(image)

    return reversed(pyramid)
```

现在看看这个金字塔的一维对齐效果。

```
shifts = np.linspace(-0.9 * ncol, 0.9 * ncol, 181)
nlevels = 8
costs = np.empty((nlevels, len(shifts)), dtype=float)
astronaut_pyramid = list(gaussian_pyramid(astronaut, levels=nlevels))
for col, shift in enumerate(shifts):
    shifted = ndi.shift(astronaut, (0, shift))
    shifted_pyramid = gaussian_pyramid(shifted, levels=nlevels)
    for row, image in enumerate(shifted_pyramid):
        costs[row, col] = mse(astronaut_pyramid[row], image)

fig, ax = plt.subplots()
for level, cost in enumerate(costs):
    ax.plot(shifts, cost, label='Level %d' % (nlevels - level))
ax.legend(loc='lower right', frameon=True, framealpha=0.9)
```

```
ax.set_xlabel('Shift')
ax.set_ylabel('MSE');
```

如你所见，金字塔的最高级在约 −325 像素处的隆起消失了。因此，可以在这一级上得到一个近似的对齐结果，然后再用较低级别改善对齐结果（见图 7-2）。

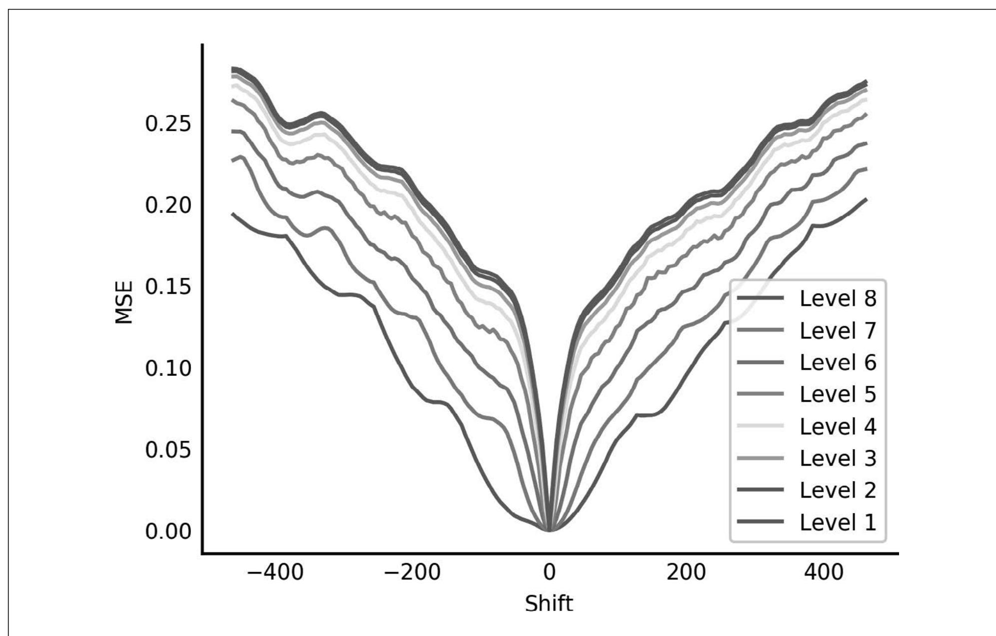


图 7-2：高斯金字塔各级别上的平移均方误差

7.2 用optimize进行图像配准

接下来我们将这个优化过程自动化，并用 3 个参数实现“真正的”对齐，这 3 个参数是旋转、行维度上的翻译和列维度上的翻译。这种操作称为“**刚性**配准”，因为不涉及任何图像变形操作（缩放、扭曲或拉伸）。目标图像被看作固体，可以四处移动（包括旋转），直至找到一个匹配。

为了简化代码，我们将用 scikit-image 中的 `transform` 模块来计算图像的平移和旋转。SciPy 的 `optimize` 模块要求输入是参数向量，我们先建立一个函数，它能接受这种向量并用正确的参数生成刚性转换。

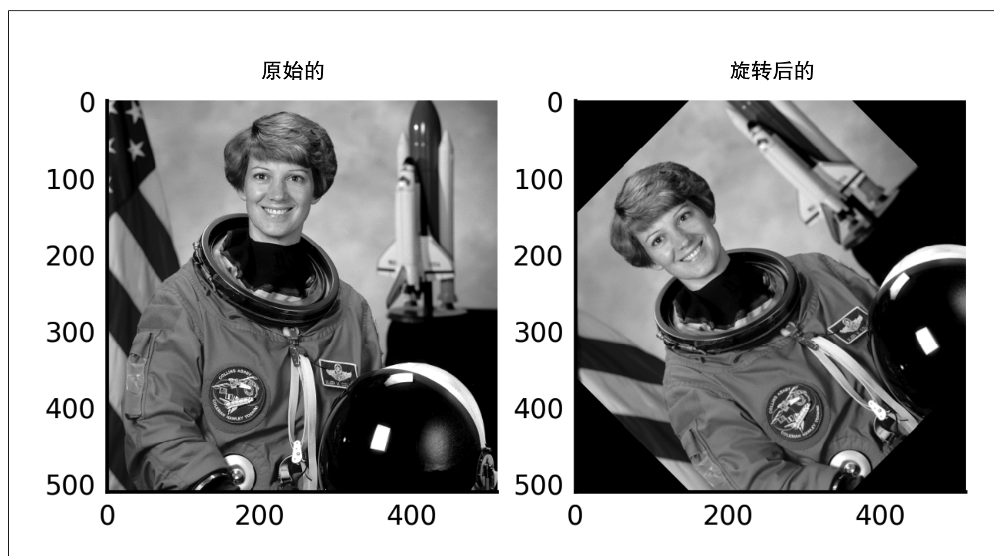
```
from skimage import transform

def make_rigid_transform(param):
    r, tc, tr = param
    return transform.SimilarityTransform(rotation=r,
                                         translation=(tc, tr))
rotated = transform.rotate(astronaut, 45)
```

```
fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].imshow(astronaut)
axes[0].set_title('Original')
axes[1].imshow(rotated)
axes[1].set_title('Rotated');
```

下一步需要一个损失函数。损失函数就是 MSE，但 SciPy 要求特定格式：第一个参数必须是用来优化的**参数向量**。其后的参数可以作为元组通过 `args` 关键字来传递，但必须是不可变的，即只有参数向量能进行优化。在我们的示例中，参数向量就是旋转角度和两个翻译参数。

```
def cost_mse(param, reference_image, target_image):
    transformation = make_rigid_transform(param)
    transformed = transform.warp(target_image, transformation, order=3)
    return mse(reference_image, transformed)
```



最后，要完成对齐函数，它**在高斯金字塔的每一级上**优化损失函数，使用前一级别的结果作为下一级别的起始点。

```
def align(reference, target, cost=cost_mse):
    nlevels = 7
    pyramid_ref = gaussian_pyramid(reference, levels=nlevels)
    pyramid_tgt = gaussian_pyramid(target, levels=nlevels)

    levels = range(nlevels, 0, -1)
    image_pairs = zip(pyramid_ref, pyramid_tgt)
    p = np.zeros(3)

    for n, (ref, tgt) in zip(levels, image_pairs):
        p[1:] *= 2

        res = optimize.minimize(cost, p, args=(ref, tgt), method='Powell')
        p = res.x
```

```
        # 输出当前级别，每次覆盖上一次的输出（就像进度条）
        print(f'Level: {n}, Angle: {np.rad2deg(res.x[0]) :.3}, '
              f'Offset: ({res.x[1] * 2**n :.3}, {res.x[2] * 2**n :.3}), '
              f'Cost: {res.fun :.3}', end='\r')

    print('') # 对齐完成后开始新一行
    return make_rigid_transform(p)
```

接下来用宇航员的图片测试一下。将这张图片旋转 60 度，并添加一些噪声。SciPy 能找到正确的转换形式吗（见图 7-3）？

```
from skimage import util

theta = 60
rotated = transform.rotate(astronaut, theta)
rotated = util.random_noise(rotated, mode='gaussian',
                            seed=0, mean=0, var=1e-3)

tf = align(astronaut, rotated)
corrected = transform.warp(rotated, tf, order=3)

f, (ax0, ax1, ax2) = plt.subplots(1, 3)
ax0.imshow(astronaut)
ax0.set_title('Original')
ax1.imshow(rotated)
ax1.set_title('Rotated')
ax2.imshow(corrected)
ax2.set_title('Registered')
for ax in (ax0, ax1, ax2):
    ax.axis('off')

Level: 1, Angle: -60.0, Offset: (-1.87e+02, 6.98e+02), Cost: 0.0369
```



图 7-3：用优化方法对齐图像

现在感觉非常好，但参数的选择掩盖了优化方法的难点。来看一下旋转 50 度会是什么情况，这与初始图像**更加接近**。

```python
theta = 50
rotated = transform.rotate(astronaut, theta)
rotated = util.random_noise(rotated, mode='gaussian',
                            seed=0, mean=0, var=1e-3)

tf = align(astronaut, rotated)
corrected = transform.warp(rotated, tf, order=3)

f, (ax0, ax1, ax2) = plt.subplots(1, 3)
ax0.imshow(astronaut)
ax0.set_title('Original')
ax1.imshow(rotated)
ax1.set_title('Rotated')
ax2.imshow(corrected)
ax2.set_title('Registered')
for ax in (ax0, ax1, ax2):
    ax.axis('off')
```

```
Level: 1, Angle: 0.414, Offset: (2.85, 38.4), Cost: 0.141
```

虽然这次从与初始图像更接近的地方开始，但图像并没有被正确地旋转过来（见图 7-4）。这是因为优化技术会掉入局部最小值的陷阱，我们已经在前面平移的对齐过程中见识过了，这是通往成功的一点小问题，但它们对初始参数非常敏感。



图 7-4：失败的优化

7.3 用basin hopping算法避开局部最小值

basin hopping 是 David Wales 和 Jonathan Doyle 在 1997 年设计的一种算法 [2]，旨在避开局部最小值。它先根据某些初始参数进行优化，然后从找到的局部最小值向随机方向移动，然后再进行优化。通过为这些随机移动选择合适的步长，这种算法可以避开两次求出同一个局部最小值的情况，因此，与基于梯度的简单优化方法相比，它可以探索更大范围的参数空间。

注 2：WALES D, DOYLE J. Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms [J]. Journal of Physical Chemistry, 1997, 101(28): 5111-5116.

留给读者一项练习：在对齐函数中加入 SciPy 对 basin hopping 算法的实现。因为本章后面的内容会用到这个算法，所以如果遇到困难的话，你完全可以翻看一下本书附录部分的解决方案。

练习：修改对齐函数

修改 `align` 函数以使用 `scipy.optimize.basinhopping`，它有避开局部最小值的明确策略。



basin hopping 算法应限制于高斯金字塔的最高级别，因为它是一个速度非常慢的优化方法。如果用于整个解决方案，那么运行时间就会非常长。

7.4 选择正确的目标函数

现在我们有了一套行之有效的图像配准方法，效果也非常好。但事实证明，我们只解决了最简单的图像配准问题：对齐**模态**相同的图像。也就是说，我们期待基准图片中的明亮像素匹配测试图片中的明亮像素。

接下来研究一下如何对齐同一图像中的不同颜色通道。这时就不能再指望这些通道具有同样的模态了。这个任务是有历史意义的。1909—1915 年，摄影家 Sergei Mikhailovich Prokudin-Gorskii 在彩色摄影技术发明前就拍摄了很多有关俄罗斯的彩色照片。他的做法是对同一场景拍摄 3 张不同的单色照片，每次拍摄都在镜头前放一张过滤不同颜色的滤色片。

这种情况下，对齐明亮像素的方法（就像 MSE 隐式实现的那样）就失效了。以下示例是圣约翰教堂污迹斑斑的 3 张玻璃窗照片，来自美国国会图书馆 Prokudin-Gorskii 作品集（见图 7-5）。

```
from skimage import io
stained_glass = io.imread('data/00998v.jpg') / 255 # 使用[0, 1]范围内的浮点数图像
fig, ax = plt.subplots(figsize=(4.8, 7))
ax.imshow(stained_glass)
ax.axis('off');
```

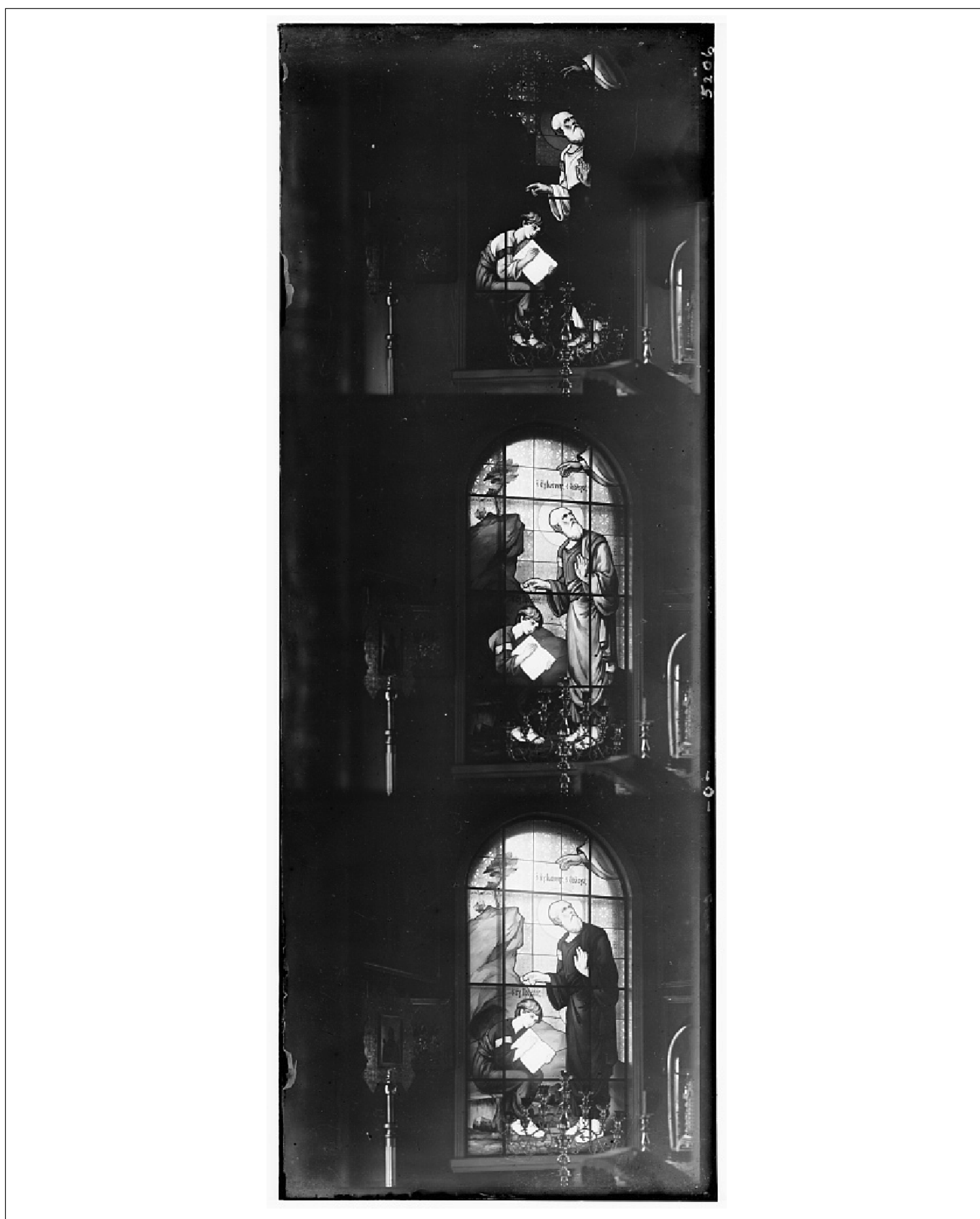
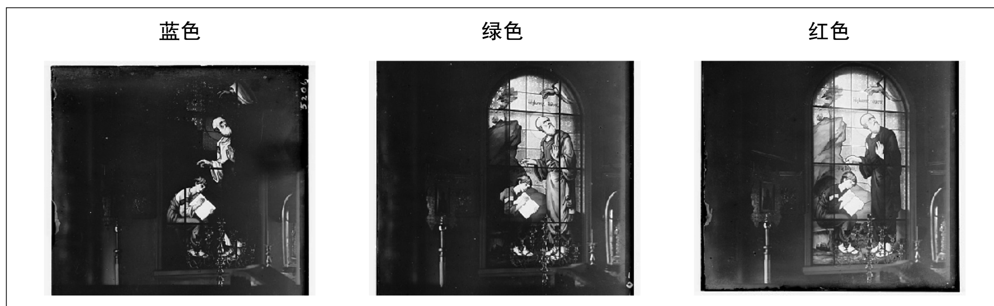


图 7-5：一张 Prokudin-Gorskii 的摄影底片：污迹斑斑的玻璃窗的 3 张照片，分别通过 3 张不同的滤色片拍摄

观察圣约翰的袍子：第一张图片中，它看上去漆黑一片；第二张图片中却是灰色的；第三张图片中竟然是亮白色！即使完美对齐，这也会导致可怕的 MSE 得分。

接下来看看能做些什么。作为起点，我们将这张底片按照成分通道进行分割。

```
nrows = stained_glass.shape[0]
step = nrows // 3
channels = (stained_glass[:step],
            stained_glass[step:2*step],
            stained_glass[2*step:3*step])
channel_names = ['blue', 'green', 'red']
fig, axes = plt.subplots(1, 3)
for ax, image, name in zip(axes, channels, channel_names):
    ax.imshow(image)
    ax.axis('off')
    ax.set_title(name)
```



第一步，我们将 3 张图片叠加起来，检验一下是否需要在 3 个通道之间调整对齐方式。

```
blue, green, red = channels
original = np.dstack((red, green, blue))
fig, ax = plt.subplots(figsize=(4.8, 4.8), tight_layout=True)
ax.imshow(original)
ax.axis('off');
```



从图中物体周围的彩色“光晕”可以看出，颜色接近于对齐，但还不够精确。就像对齐宇航员图片那样，我们试着用 MSE 来对齐它们。我们将绿色通道图片作为基准图片，让蓝色通道图片和红色通道图片与其对齐。

```
print('*** Aligning blue to green ***')
tf = align(green, blue)
cblue = transform.warp(blue, tf, order=3)

print('** Aligning red to green ***')
tf = align(green, red)
cred = transform.warp(red, tf, order=3)

corrected = np.dstack((cred, green, cblue))
f, (ax0, ax1) = plt.subplots(1, 2)
ax0.imshow(original)
ax0.set_title('Original')
ax1.imshow(corrected)
ax1.set_title('Corrected')
for ax in (ax0, ax1):
    ax.axis('off')

*** Aligning blue to green ***
Level: 1, Angle: -0.0474, Offset: (-0.867, 15.4), Cost: 0.0499
** Aligning red to green ***
Level: 1, Angle: 0.0339, Offset: (-0.269, -8.88), Cost: 0.0311
```

对齐后的图片效果要比原始图片好一些（见图 7-6），因为红色通道和绿色通道正确对齐了，这也许要归功于那一大片黄色天空背景。但蓝色通道还没有对齐，因为蓝色亮点和绿色通道并不一致。这说明，当通道**并未**对齐时，MSE 会变得更低，因此蓝色区域会与一些明亮的绿色区域重叠。



图 7-6：基于 MSE 的对齐可以减少但不能消除色晕

我们使用另一个称为**标准化互信息**（NMI，normalized mutual information）的指标，它度量的是不同图片的不同亮度带之间的相关性。当图片完美对齐时，颜色一致的任何物体都

会在不同成分通道的遮罩之间产生很大的相关性，并相应产生一个很大的 NMI 值。从某种意义上说，NMI 测量的是，在给定一张图片像素值的情况下预测另一张图片中相应像素值的容易程度。其定义如下：[3]

$$I(X,Y)=\frac{H(X)+H(Y)}{H(X,Y)}$$

其中 $H(X)$ 是 X 的**熵**，$H(X, Y)$ 是 X 和 Y 的联合熵。分子表示两张图片各自的熵，分母表示它们合在一起时总的熵，它的值在 1（最大程度对齐）和 2（最小程度对齐）之间。[4] 参见第 5 章。

Python 代码如下所示。

```
from scipy.stats import entropy

def normalized_mutual_information(A, B):
    """Compute the normalized mutual information.

    The normalized mutual information is given by:

                 H(A) + H(B)
      Y(A, B) = -----------
                  H(A, B)

    where H(X) is the entropy ``- sum(x log x) for x in X``.

    Parameters
    ----------
    A, B : ndarray
        Images to be registered.

    Returns
    -------
    nmi : float
        The normalized mutual information between the two arrays, computed at a
        granularity of 100 bins per axis (10,000 bins total).
    """
    hist, bin_edges = np.histogramdd([np.ravel(A), np.ravel(B)], bins=100)
    hist /= np.sum(hist)

    H_A = entropy(np.sum(hist, axis=0))
    H_B = entropy(np.sum(hist, axis=1))
    H_AB = entropy(np.ravel(hist))

    return (H_A + H_B) / H_AB
```

注 3：STUDHOLME C, HILL D L G, HAWKES D J. An overlap invariant entropy measure of 3D medical image alignment [J]. Pattern Recognition, 1999, 32(1): 71-86.

注 4：另一种煞有介事的简单解释就是，可以根据要考虑的量的直方图来计算熵。如果 $X = Y$，那么联合直方图 (X, Y) 就是对角阵，而且这个对角阵的熵和 X 或 Y 的熵都相同。因此，$H(X) = H(Y) = H(X, Y)$，$I(X, Y) = 2$。

和前面定义 `cost_mse` 一样，我们需要定义一个**损失函数**以进行优化。

```
def cost_nmi(param, reference_image, target_image):
    transformation = make_rigid_transform(param)
    transformed = transform.warp(target_image, transformation, order=3)
    return -normalized_mutual_information(reference_image, transformed)
```

最后，通过带有 basin hopping 优化方法的对齐函数来优化损失函数（见图 7-7）。

```
print('*** Aligning blue to green ***')
tf = align(green, blue, cost=cost_nmi)
cblue = transform.warp(blue, tf, order=3)

print('** Aligning red to green ***')
tf = align(green, red, cost=cost_nmi)
cred = transform.warp(red, tf, order=3)

corrected = np.dstack((cred, green, cblue))
fig, ax = plt.subplots(figsize=(4.8, 4.8), tight_layout=True)
ax.imshow(corrected)
ax.axis('off')

*** Aligning blue to green ***
Level: 1, Angle: 0.444, Offset: (6.07, 0.354), Cost: -1.08
** Aligning red to green ***
Level: 1, Angle: 0.000657, Offset: (-0.635, -7.67), Cost: -1.11

(-0.5, 393.5, 340.5, -0.5)
```



图 7-7：用标准化互信息对齐 Prokudin-Gorskii 图片中的通道

多么绚丽的一张照片！这可是彩色照相技术发明前创造出的人工作品！看看上帝那像珍珠一样的白色锦袍、约翰的白胡子以及伯罗哥罗（Prochorus，约翰的书童）手中的白色书页，这些都是无法在基于 MSE 的对齐中看到的，但使用 NMI 后，它们显得栩栩如生。连前面的烛台看上去都那么金碧辉煌。

本章介绍了函数优化的两个核心思想：理解局部最小值以及如何避开它们，选择正确的函数优化以达到特定目标。如果能解决这些问题，你就可以优化很多科学问题了！

第 8 章

用Toolz在笔记本电脑上玩转大数据

格蕾丝："一把刀？那家伙有 12 英尺高！"

杰克："7 英尺。嘿，别担心，我能搞定他。"[1]

——杰克·波顿，《妖魔大闹唐人街》

流（streaming）不是 SciPy 本身的功能，而是一种可以让我们高效处理大数据集的方法，比如科学研究中常见的大数据集。Python 语言中有一些适合流数据处理的基本功能，这些功能和 Matt Rocklin 开发的 Toolz 库相结合，可以写出非常优雅、简洁的代码，并且极其节省内存。本章将介绍如何应用流的思想来处理那些超过计算机内存的大规模数据集。

你很可能已经做过一些流处理，但也许没有从这些方面思考过流。最简单的流处理形式就是在文件的行之间迭代，不用把整个文件读入内存，就能处理每一行。例如，以下代码用一个循环来计算每行的均值，并对其进行加总。

```
import numpy as np
with open('data/expr.tsv') as f:
    sum_of_means = 0
    for line in f:
        sum_of_means += np.mean(np.fromstring(line, dtype=int, sep='\t'))
print(sum_of_means)
```

```
1463.0
```

在通过行处理就能解决问题的情况下，这种策略效果良好，但当代码变得更加复杂时，情况很快就会失控。

在流处理程序中，一个函数先处理**某些**输入数据，并返回处理结果。然后，当下游函数处

注 1：1 英尺约合 30.48 厘米。——编者注

理这个结果时，这个函数再接受一些数据进行处理，以此类推。这些事情都是同时进行的！如何保持它们井然有序地进行呢？

我们也觉得这个事情很难，直到发现了 Toolz 库，其结构使得流处理程序编写起来特别优雅，因此本书必须用一章的篇幅来介绍它。

我们先阐明“流”的概念，以及为什么要使用流。假设你有一些保存在文本文件中的数据，你想计算出对每个值进行 $\log(x + 1)$ 运算后的列平均值。通常的做法是先用 NumPy 加载这些数据，再对整个矩阵中的值进行对数运算，然后沿着第一个轴求出均值。

```
import numpy as np
expr = np.loadtxt('data/expr.tsv')
logexpr = np.log(expr + 1)
np.mean(logexpr, axis=0)

array([ 3.11797294, 2.48682887, 2.19580049, 2.36001866, 2.70124539,
        2.64721531, 2.43704834, 3.28539133, 2.05363724, 2.37151577,
        3.85450782, 3.9488385 , 2.46680157, 2.36334423, 3.18381635,
        2.64438124, 2.62966516, 2.84790568, 2.61691451, 4.12513405])
```

这可以奏效，而且非常符合我们已经得心应手的输入－输出计算模型。但这样做非常低效！我们在内存中载入了整个矩阵（1），然后对每个元素加 1 复制了一个矩阵（2），之后为了计算对数又复制了一个矩阵（3），最后才将这个矩阵传给 `np.mean`。这样数据数组就有了 3 个实例，实际上这种计算甚至连**一个**内存实例都不需要。对于任何一种“大数据”操作，这种方法都会无效。

Python 开发者意识到了这个问题，于是创建了 `yield` 关键字，它可以使函数先处理“一份”数据，将结果传递给下面的进程，**待完成对这份数据的一系列处理后**，再处理另一份数据。“yield”非常恰当地描述了这个过程：函数将控制权“移交”给下一个函数，直到后续的数据处理步骤都完成后，才继续处理接下来的数据。

8.1 用yield进行流处理

以上所说的控制流非常难以把握。Python 中一项非常好的功能就是对这种控制复杂性进行了抽象，从而使你更专注于功能分析。可以这样理解：如果一个函数在正常情况下接受一个列表（数据集合）并将其转换，那么可以重写这个函数，使它接受一个**流**，并**依次**对流中的每个元素生成一个结果。

以下示例对列表中的每个元素取对数，分别使用了标准的数据复制方法和流处理方法。

```
def log_all_standard(input):
    output = []
    for elem in input:
        output.append(np.log(elem))
    return output

def log_all_streaming(input_stream):
    for elem in input_stream:
        yield np.log(elem)
```

检查两种方法的结果是否一致。

```
# 设定随机数种子，以得到稳定的结果
np.random.seed(seed=7)
# 设置输出选项，只显示3位有效数字
np.set_printoptions(precision=3, suppress=True)

arr = np.random.rand(1000) + 0.5
result_batch = sum(log_all_standard(arr))
print('Batch result: ', result_batch)
result_stream = sum(log_all_streaming(arr))
print('Stream result: ', result_stream)

Batch result:  -48.2409194561
Stream result:  -48.2409194561
```

流处理的优点是，只在需要时才处理流中的元素，无论是累计求和、写入磁盘，还是其他什么操作。这样，当输入项目很多或每个输入项很大（或二者皆有）时，可以节省大量内存。以下是引自 Matthew Rocklin 的博文“Towards Out-of-core ND-Arrays —— Dask + Toolz = Bag”中的一段话，这段话简明扼要地解释了流式数据分析的好处：

> 根据我的有限经验，人们很少使用这种（流）方法，他们会一直使用单线程的驻留内存的 Python 程序，直到系统崩溃，然后就去购买那些费用高昂的大数据基础设施，比如 Hadoop/Spark。

的确，这就是我们计算生涯的真实写照。但是，还有一条效果**超乎想象**的中间道路。在某些情况下，通过消除多核通信和随机访问数据库的开销，它的速度甚至比超级计算方法还要快。（例如，参见 Frank McSherry 的博文“Bigger data; same laptop”，他在笔记本电脑上处理了一个带有 1280 亿条边的图，速度比使用超级计算机上的图数据库**更快**。）

为了阐释清楚流处理风格函数的控制流，有必要看一下函数的**详细**运行信息，其中包含了每次操作的输出消息。

```
import numpy as np

def tsv_line_to_array(line):
    lst = [float(elem) for elem in line.rstrip().split('\t')]
    return np.array(lst)

def readtsv(filename):
    print('starting readtsv')
    with open(filename) as fin:
        for i, line in enumerate(fin):
            print(f'reading line {i}')
            yield tsv_line_to_array(line)
    print('finished readtsv')

def add1(arrays_iter):
    print('starting adding 1')
    for i, arr in enumerate(arrays_iter):
        print(f'adding 1 to line {i}')
        yield arr + 1
```

```
    print('finished adding 1')

def log(arrays_iter):
    print('starting log')
    for i, arr in enumerate(arrays_iter):
        print(f'taking log of array {i}')
        yield np.log(arr)
    print('finished log')

def running_mean(arrays_iter):
    print('starting running mean')
    for i, arr in enumerate(arrays_iter):
        if i == 0:
            mean = arr
        mean += (arr - mean) / (i + 1)
        print(f'adding line {i} to the running mean')
    print('returning mean')
    return mean
```

来看一下在一个小示例文件上的运行结果。

```
fin = 'data/expr.tsv'
print('Creating lines iterator')
lines = readtsv(fin)
print('Creating loglines iterator')
loglines = log(add1(lines))
print('Computing mean')
mean = running_mean(loglines)
print(f'the mean log-row is: {mean}')

Creating lines iterator
Creating loglines iterator
Computing mean
starting running mean
starting log
starting adding 1
starting readtsv
reading line 0
adding 1 to line 0
taking log of array 0
adding line 0 to the running mean
reading line 1
adding 1 to line 1
taking log of array 1
adding line 1 to the running mean
reading line 2
adding 1 to line 2
taking log of array 2
adding line 2 to the running mean
reading line 3
adding 1 to line 3
taking log of array 3
adding line 3 to the running mean
reading line 4
adding 1 to line 4
```

```
taking log of array 4
adding line 4 to the running mean
finished readtsv
finished adding 1
finished log
returning mean
the mean log-row is: [ 3.118 2.487 2.196 2.36 2.701 2.647 2.437 3.285
                       2.054 2.372
  3.855  3.949  2.467  2.363 3.184 2.644 2.63 2.848 2.617 4.125]
```

注意以下事项。

- 创建行和对数行的迭代器时，没有进行任何计算。这是因为迭代器是**惰性的**，它们直到需要一个结果时才进行求值（或**消费**）。
- 当最终通过调用 `running_mean` 触发计算时，计算会在各个函数之间来回切换，在每一行上执行各种计算，然后转到下一行。

8.2　引入Toolz流库

本章示例代码由 Matthew Rocklin 提供。只需几行代码，就可以在一台笔记本电脑上根据完整的果蝇基因组在 5 分钟内建立一个马尔可夫模型。（为了便于下游处理，代码略加修改。）Matthew 的示例使用的是人类基因组，但笔记本电脑速度显然不够快，因此我们使用果蝇基因组（大小约是人类基因组的 1/20）。本章还对代码做了一点增强，使其可以从压缩数据开始（谁会将未压缩的数据集放在硬盘上呢）。这种修改对示例代码的优雅程度**几乎没有影响**。

```
import toolz as tz
from toolz import curried as c
from glob import glob
import itertools as it

LDICT = dict(zip('ACGTacgt', range(8)))
PDICT = {(a, b): (LDICT[a], LDICT[b])
         for a, b in it.product(LDICT, LDICT)}

def is_sequence(line):
    return not line.startswith('>')

def is_nucleotide(letter):
    return letter in LDICT # 忽略N

@tz.curry
def increment_model(model, index):
    model[index] += 1


def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted, # 文件名
                   c.map(open), # 行
                   # 连接所有文件中的行
```

```
                    tz.concat,
                    # 去掉每个序列的标题
                    c.filter(is_sequence),
                    # 连接所有行中的字符
                    tz.concat,
                    # 去掉换行符和N
                    c.filter(is_nucleotide))


def markov(seq):
    """Get a 1st-order Markov model from a sequence of nucleotides."""
    model = np.zeros((8, 8))
    tz.last(tz.pipe(seq,
                    c.sliding_window(2),        # 每个连续元组
                    c.map(PDICT.__getitem__),   # 元组在矩阵中的位置
                    c.map(increment_model(model))))  # 矩阵相应元素加1
    # 将计数转换为转移概率矩阵
    model /= np.sum(model, axis=1)[:, np.newaxis]
    return model
```

可以用以下代码得到一个果蝇基因组中重复序列的马尔可夫模型。

```
%%timeit -r 1 -n 1
dm = 'data/dm6.fa'
model = tz.pipe(dm, genome, c.take(10**7), markov)
# 为了加快速度，使用take，只在前1000万个碱基上运行
# 如果你可以等5~10分钟，那么可以去掉take这一步
1 loop, average of 1: 24.3 s +- 0 ns per loop (using standard deviation)
```

这个示例中有**很多**知识点，我们会慢慢解释。本章末尾将实际运行这个示例。

需要注意的第一件事就是有多少函数来自 Toolz 库。举例来说，我们从 Toolz 中使用了 `pipe`、`sliding_window`、`frequencies` 和一个 `map` 函数的柯里化版本（后面会有更多介绍）。这是因为，Toolz 就是专门为了利用 Python 的迭代器，并使流操作更加简单而开发的。

先从 `pipe` 开始。这个函数就是一个能使嵌套函数调用更易于阅读的语法糖。因为这种模式在处理迭代器时会使用得越来越普遍，所以非常重要。

作为一个简单示例，我们用 `pipe` 重写一下求移动平均值的函数。

```
import toolz as tz
filename = 'data/expr.tsv'
mean = tz.pipe(filename, readtsv, add1, log, running_mean)

# 这种写法等价于以下嵌套函数
# running_mean(log(add1(readtsv(filename))))

starting running mean
starting log
starting adding 1
starting readtsv
reading line 0
adding 1 to line 0
taking log of array 0
```

```
adding line 0 to the running mean
reading line 1
adding 1 to line 1
taking log of array 1
adding line 1 to the running mean
reading line 2
adding 1 to line 2
taking log of array 2
adding line 2 to the running mean
reading line 3
adding 1 to line 3
taking log of array 3
adding line 3 to the running mean
reading line 4
adding 1 to line 4
taking log of array 4
adding line 4 to the running mean
finished readtsv
finished adding 1
finished log
returning mean
```

原来那种很多行的代码或乱七八糟的括号，现在都变成了对输入数据按次序转换的清晰描述，这就容易理解多了！

与最初的 NumPy 实现相比，这种方法还有一个优点。在 NumPy 实现中，如果数据规模增加到几百万甚至几十亿行，那么计算机很难将所有数据都装入内存。相比之下，这种方法每次只从磁盘载入一行数据，内存中也只保留一行数据。

8.3　*k*-mer计数与错误修正

你可能想回顾一下第 1 章和第 2 章中关于 DNA 和基因组的知识。简而言之，基因信息（也就是你的身体细胞的蓝本）被编码为**基因组**中的化学**碱基**序列。这些序列非常非常小，无法用显微镜查看，也不能读取。你也无法读取长串序列，因为累积的误差会使结果变得无法使用。（新的技术正在改变这种情况，但这里只关注那些短的可读序列，这是现在最常见的。）好在我们的每个细胞中都有基因组的一份相同副本，因此可以将这些副本打碎成很多短的序列（约 100 个碱基那么长），然后再将它们组装起来，就像组装一个 3000 万块的巨大拼图。

组装之前，至关重要的一个事情就是 read 修正。在 DNA 测序过程中，一些碱基被错误地读取了，因此必须将其修正，否则就会搞砸组装过程。（可以想象一下拼图时碎片形状不吻合的情况。）

其中一种修正策略是在数据集中找出相似的 read，从这些 read 中抽取出正确信息以修正错误，或者彻底丢弃那些包含错误的 read。

然而，这种方法非常低效，因为找出相似的 read 就意味着要将每个 read 都和其他 read 比较一次。这需要 N^2 次操作，对于一个包含 3000 万 read 的数据集来说，就是 9×10^{14} 次操作！（这些操作的开销太大了。）

还有另外一种方法。根据 Pavel Pevzner 等人的研究（参见 Pavel A. Pevzner、Haixu Tang 以及 Michael S. Waterman 的论文“An Eulerian path approach to DNA fragment assembly”），可以将 read 打碎成更小的、有重叠的 *k*-mer，即长度为 *k* 的子串，它们可以保存在一个散列表（Python 中的字典）中。这样做的好处很多，最主要的是不用计算 read 的总数，这个总数可以是任意大的值。于是，计算 *k*-mer 的总数就可以了，这个总数和基因组本身一样大——通常比 read 总数小一两个数量级。

如果我们选择了足够大的 *k* 值，确保任何 *k*-mer 都只在基因组中出现一次，那么一个 *k*-mer 的出现次数就恰好是来自那部分基因组的 read 数量。这称为该区域的**覆盖度**。

如果一个 read 中有错误，那么就很可能出现这种情况：与错误重叠的 *k*-mer 在基因组中是唯一的或接近唯一的。考虑一下英语中同样的情况：如果你从莎士比亚作品中得到 read，有个 read 是“to be or nob to be”，那么 6-mer“nob to”会很少或根本不出现，而“not to”会出现得非常频繁。

这就是 *k*-mer 错误修正方法的基础：先将 read 分割成 *k*-mer，再计算出每个 *k*-mer 的出现次数，然后通过某种逻辑用常见的 *k*-mer 替换 read 中非常罕见的相似 *k*-mer。（或者丢弃那些带有错误 *k*-mer 的 read。这是可行的，因为 read 的冗余很高，所以允许我们丢弃错误数据。）

这也是一个**必须**使用流的示例，前面说过，read 的数量非常多，我们不想将它们都保存在内存中。

DNA 序列数据通常表示为 FASTA 格式。这是一种纯文本格式，每个文件包含一个或多个 DNA 序列，由名称和实际序列组成。

以下是一个 FASTA 文件示例。

```
> sequence_name1
TCAATCTCTTTTATATTAGATCTCGTTAAAGTAAAATTTTGGTTTGTGTTAAAGTACAAG
GGGTACCTATGACCACGGAACCAACAAAGTGCCTAAATAGGACATCAAGTAACTAGCGGT
ACGT

> sequence_name2
ATGTCCCAGGCGTTCCTTTTGCATTTGCTTCGCATTAACAGAATATCCAGCGTACTTAGG
ATTGTCGACCTGTCTTGTCGTACGTGGCCGCAACACCAGGTATAGTGCCAATACAAGTCA
GACTAAAACTGGTTC
```

现在我们已经具备所需信息，FASTA 文件中的行就是一个流，我们可以将这个流转换为 *k*-mer 计数：

- 过滤行，只使用序列行；
- 为每一行生成一个 *k*-mer 流；
- 将每个 *k*-mer 加入一个字典计数器。

下面是只使用内置函数的纯 Python 代码的实现。

```
def is_sequence(line):
    line = line.rstrip() # 删除行尾的\n
    return len(line) > 0 and not line.startswith('>')

def reads_to_kmers(reads_iter, k=7):
     for read in reads_iter:
         for start in range(0, len(read) - k):
             yield read[start : start + k] # 注意yield，这是一个生成器

def kmer_counter(kmer_iter):
    counts = {}
    for kmer in kmer_iter:
      if kmer not in counts:
          counts[kmer] = 0
      counts[kmer] += 1
  return counts

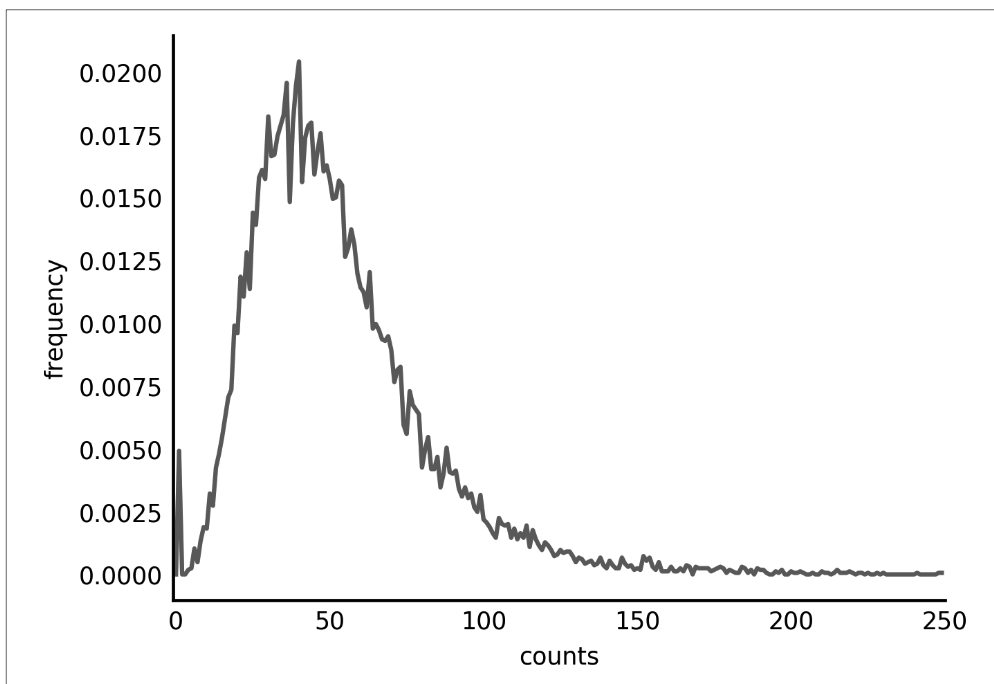
with open('data/sample.fasta') as fin:
    reads = filter(is_sequence, fin)
    kmers = reads_to_kmers(reads)
    counts = kmer_counter(kmers)
```

以上代码完全没有问题，而且以流方式工作，read 从磁盘中一个接一个地载入，依次通过 *k*-mer 转换器到达 *k*-mer 计数器。接下来可以绘制出计数的频率分布图，而且在图中确实可以看到，正确的 *k*-mer 和错误的 *k*-mer 分成了截然不同的两组。

```
# 使图形显示在文本中，定制绘图风格
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('style/elegant.mplstyle')

def integer_histogram(counts, normed=True, xlim=[], ylim=[],
                      *args, **kwargs):
    hist = np.bincount(counts)
    if normed:
        hist = hist / np.sum(hist)
    fig, ax = plt.subplots()
    ax.plot(np.arange(hist.size), hist, *args, **kwargs)
    ax.set_xlabel('counts')
    ax.set_ylabel('frequency')
    ax.set_xlim(*xlim)
    ax.set_ylim(*ylim)

counts_arr = np.fromiter(counts.values(), dtype=int, count=len(counts))
integer_histogram(counts_arr, xlim=(-1, 250))
```



从上图可以看到 *k*-mer 频率分布得非常漂亮，图形左侧有一个大的突起，只出现了一次，这种低频率的 *k*-mer 很可能就是错误。

但是在前面的代码中，我们其实做了太多工作。我们写在 `for` 循环里的功能和 `yield` 语句实际上就是**流操作**：将一种数据流转换为另一种数据，并最终累积起来。Toolz 中有很多基本的流操作功能，可以很轻松地将以上代码的功能在一次函数调用中实现。而且一旦知道了转换函数的名称，数据流中每一点变化的可视化也变得非常容易。

举例来说，**滑动窗口**函数可以完美地满足我们生成 *k*-mer 的需要。

```
print(tz.sliding_window.__doc__)

 A sequence of overlapping subsequences

    >>> list(sliding_window(2, [1, 2, 3, 4]))
    [(1, 2), (2, 3), (3, 4)]

    This function creates a sliding window suitable for transformations like
    sliding means / smoothing

    >>> mean = lambda seq: float(sum(seq)) / len(seq)
    >>> list(map(mean, sliding_window(2, [1, 2, 3, 4])))
    [1.5, 2.5, 3.5]
```

此外，**频率**函数可以为数据流中的独立项目计数。结合管道操作，可以在一次函数调用中完成 *k*-mer 计数。

```
from toolz import curried as c

k = 7
counts = tz.pipe('data/sample.fasta', open,
                 c.filter(is_sequence),
                 c.map(str.rstrip),
                 c.map(c.sliding_window(k)),
                 tz.concat, c.map(''.join),
                 tz.frequencies)
```

不过先等等，那些来自 `toolz.curried` 的 `c.function` 到底是什么呢？

8.4　柯里化：流的调料

前面我们使用了 `map` 函数的**柯里化**版本，它可以将一个给定函数应用到序列中的每个元素。既然已经使用了很多柯里化的函数，现在就来解释一下什么是柯里化。柯里化不是根据那种调料命名的（尽管它可以使代码的味道更好），[2] 它的名称来自提出这个概念的数学家 Haskell Curry。以 Haskell Curry 命名的还有 Haskell 编程语言，其中的**所有**函数都是柯里化的！

“柯里化”意味着对函数进行**部分**求值，并将未求值的部分作为“更小”的函数返回。通常来说，在 Python 中，如果没有给函数提供所需的所有参数，那么函数就会抛出错误。与此不同，柯里化的函数可以接受**部分**参数。如果没有得到足够的参数，它会返回一个能接受剩余参数的新函数。如果用剩余参数调用第二个函数，那么它会继续执行原来的任务。表示柯里化的另一个术语就是**部分求值**（partial evaluation）。在函数式编程中，想要生成等待随后出现的其余参数的函数，柯里化是一种方法。

因此，如果函数调用 `map(np.log, numbers_list)`，对 `number_list` 中的所有数值应用 `np.log` 函数（返回一个对数值序列），那么 `toolz.curried.map(np.log)` 就返回一个**函数**，这个函数可以接受一个序列并返回取对数后的序列。

事实证明，部分参数已知的函数完美匹配流操作！在以上代码片段中，我们已经看到，柯里化和管道结合使用时是多么强大。

但刚开始使用柯里化时，它可能不太好理解，因此先用一些简单示例来演示其用法。我们先写一个简单的、非柯里化的函数。

```
def add(a, b):
    return a + b

add(2, 5)

7
```

然后编写一个手动柯里化的类似函数。

注 2：柯里化的英文是 currying，curry 是咖喱的意思。——译者注

```
def add_curried(a, b=None):
    if b is None:
        # 没有给定第二个参数，因此定义一个函数并返回该函数
        def add_partial(b):
            return add(a, b)
        return add_partial
    else:
        # 两个参数都给定了，因此可以返回一个值
        return add(a, b)
```

接着测试一下柯里化函数，确定它能按我们的期望工作。

```
add_curried(2, 5)
```

```
7
```

当给定两个变量时，它就像一个普通函数。现在留出第二个变量。

```
add_curried(2)
```

```
<function __main__.add_curried.<locals>.add_partial>
```

不出所料，它返回了一个函数。现在使用这个函数。

```
add2 = add_curried(2)
add2(5)
```

```
7
```

这样做是可以的，但 `add_curried` 是一个很难读的函数，未来我们很可能忘记是如何编写这段代码的。好在 Toolz 中的工具可以帮助我们摆脱这种困境。

```
import toolz as tz

@tz.curry # 使用柯里化作为装饰器
def add(x, y):
    return x + y

add_partial = add(2)
add_partial(5)
```

```
7
```

总结一下，`add` 现在是一个柯里化的函数，它可以接受某一个参数，并返回一个可以“记住”这个参数的函数，即 `add_partial`。

实际上，Toolz 中的所有函数在 `toolz.curried` 命名空间中都可以作为柯里化的函数。Toolz 中还有一些方便好用的高阶 Python 函数的柯里化版本，包括 `map`、`filter` 和 `reduce`。我们将 `curried` 命名空间导入为 `c`，目的是避免代码过于杂乱。因此，`map` 函数的柯里化版本就是 `c.map`。注意，柯里化函数（如 `c.map`）与 `@curry` 装饰器不同，后者用于**创建**柯里化函数。

```
from toolz import curried as c
c.map

<class 'map'>
```

提醒一下，map 是内置函数。下面的文字摘自 Python 文档：

> map(function, iterable, ...) 返回一个迭代器，它可以将 function 应用于 iterable 中的每个项目，并逐步生成结果。

柯里化的 map 函数特别适合在 Toolz 管道中使用。可以先只给 c.map 传递一个函数，然后用 tz.pipe 在迭代器中进行流操作。查看我们读取基因组数据的函数，你会看到它实际是如何应用的。

```
def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted, # 文件名
                   c.map(open), # 行
                   # 连接所有文件中的行
                   tz.concat,
                   # 去掉每个序列的标题
                   c.filter(is_sequence),
                   # 连接所有行中的字符
                   tz.concat,
                   # 去掉换行符和N
                   c.filter(is_nucleotide))
```

8.5 回到*k*-mer计数

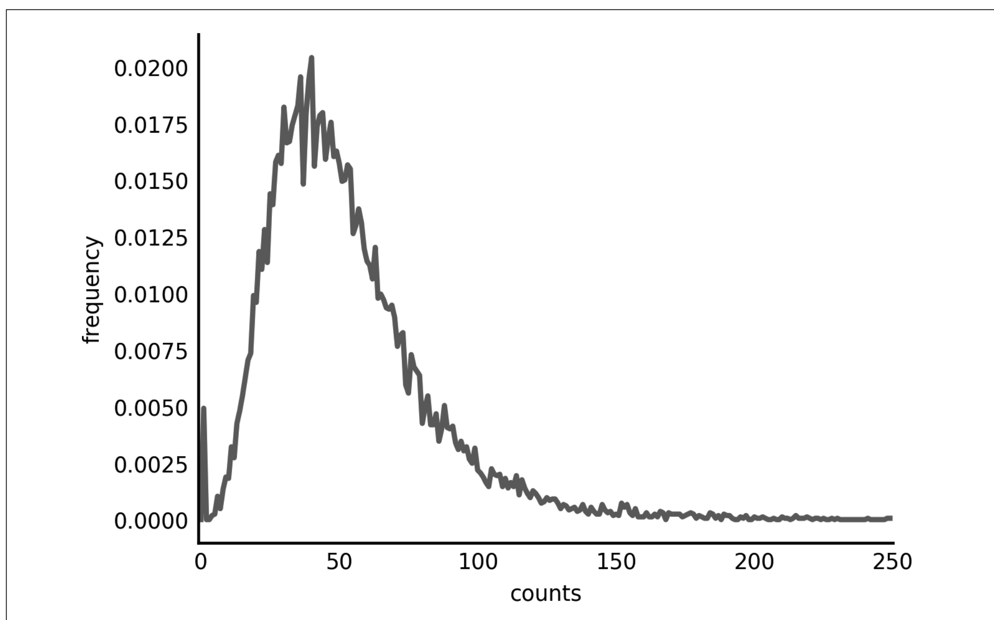
了解什么是柯里化后，让我们回到 *k*-mer 计数代码。下面是使用柯里化函数的代码。

```
from toolz import curried as c

k = 7
counts = tz.pipe('data/sample.fasta', open,
                 c.filter(is_sequence),
                 c.map(str.rstrip),
                 c.map(c.sliding_window(k)),
                 tz.concat, c.map(''.join),
                 tz.frequencies)
```

现在可以看一下不同 *k*-mer 的出现频率。

```
counts = np.fromiter(counts.values(), dtype=int, count=len(counts))
integer_histogram(counts, xlim=(-1, 250), lw=2)
```



使用流的几个小技巧

- 用 tz.concat 将“列表的列表”转换为“长列表”。
- 注意以下问题。
  - 迭代器是消耗品。如果你创建了一个生成器对象，并用它进行了一些操作，但后面的步骤出现了错误，那么你需要重新创建这个生成器。原来的生成器已经不存在了。
  - 迭代器是惰性的，有时需要强制求值。
- 当管道中有多个函数时，有时很难搞清楚哪里出了问题。此时可以使用一个小的流，从第一个（最左侧）函数开始向管道中逐个添加函数，直到找到出错的函数。你还可以在流的任意位置插入 `map(do(pring))`（`map` 和 `do` 都来自 `toolz.curried`），以打印出流中的每个元素。

练习：流数据的主成分分析

scikit-learn 库中有一个 `IncrementalPCA` 类，该类可以在不将整个数据集载入内存的情况下对数据集进行主成分分析。但你需要自己把数据分块，这就增加了编写代码的难度。编写一个能接受数据样本流并进行主成分分析的函数，然后用这个函数为 `iris` 机器学习数据集做主成分分析，这个数据集在 data/iris.csv 中。（还可以使用 `datasets.load_iris()` 函数从 scikit-learn 的 `datasets` 模块中获取这个数据集。）或者，你也可以使用种类编号为数据点着色，可以在 data/iris-target.csv 中找到种类编号。



`IncrementalPCA` 类位于 `sklearn.decomposition` 模块中，它需要一个大于 1 的**批量大小**（batch size）参数来训练模型。查看 `toolz.curried.partition` 函数，了解如何从一个数据点流创建出一个批量流。

8.6 全基因组的马尔可夫模型

回到原来的示例代码，什么是马尔可夫模型？它有什么用途？

一般来说，马尔可夫模型假设系统转移到某个特定状态的概率只依赖于其所处的前一个状态。例如，如果现在艳阳高照，那么明天很可能也是晴天，而昨天下过雨这一事实则根本不在考虑范围之内。根据这个理论，预测未来所需的所有信息都包含在事物的当前状态中，以前的状态无关紧要。对于那些难以用其他方法处理的问题，使用这个假设进行简化是非常有用的，而且经常能得到非常好的结果。例如，移动电话的信号处理和卫星通信经常会使用马尔可夫模型。

接下来我们会看到，对基因组而言，不同的基因功能区域在相似状态间具有不同的**转移概率**。在一个新基因组中观测这种现象，可以根据观测结果对这些区域进行某种预测。还是拿天气来类比，天气晴转雨的概率，在洛杉矶的情况会不同于在伦敦。因此，如果给你一系列天气状态（晴、晴、晴、雨、晴……），并假设你已经有了一个预先训练好的模型，那你就能预测这种天气是出自洛杉矶还是伦敦。

本章只涉及模型构建。

你需要下载黑腹果蝇基因组文件 dm6.fa.gz（http://hgdownload.cse.ucsc.edu/goldenPath/dm6/bigZips/），然后用 `gzip -d dm6.fa.gz` 命令来解压缩文件。

在这份基因组数据中，基因序列由字母 A、C、G 和 T 组成，按照它们是小写（重复）还是大写（非重复）编码为**重复片段**。构建马尔可夫模型时可以利用这个信息。

我们将马尔可夫模型表示为 NumPy 数组，因此要使用字典建立字母到 [0, 7] 之间标号的索引（`LDICT` 表示“字母字典”），以及字母对到（[0, 7], [0, 7]）之间的二维标号的索引（`PDICT` 表示“字母对字典”）。

```
import itertools as it

LDICT = dict(zip('ACGTacgt', range(8)))
PDICT = {(a, b): (LDICT[a], LDICT[b])
         for a, b in it.product(LDICT, LDICT)}
```

我们还要过滤无意义的数据：序列名称，即以 `>` 开头的行，还有用 `N` 标记的未知序列。通过以下函数来过滤。

```
def is_sequence(line):
    return not line.startswith('>')

def is_nucleotide(letter):
    return letter in LDICT # 忽略N
```

最后，只要得到一个新核苷酸对（如 (A, T)），就在马尔可夫模型（NumPy 矩阵）的相应位置增加一项。我们用一个柯里化的函数来完成这个操作。

```
import toolz as tz

@tz.curry
def increment_model(model, index):
    model[index] += 1
```

现在可以将这些函数组合起来，将一个基因组通过流处理转换为 NumPy 矩阵。需要注意的是，如果下面的 seq 是一个流，那么就不需要将整个基因组（甚至基因组中的一大块）保存在内存中！

```
from toolz import curried as c

def markov(seq):
    """Get a 1st-order Markov model from a sequence of nucleotides."""
    model = np.zeros((8, 8))
    tz.last(tz.pipe(seq,
                    c.sliding_window(2),        # 每个连续元组
                    c.map(PDICT.__getitem__),   # 元组在矩阵中的位置
                    c.map(increment_model(model)))) # 矩阵相应元素加1
    # 将计数转换为转移概率矩阵
    model /= np.sum(model, axis=1)[:, np.newaxis]
    return model
```

现在只需要生成基因组流，并建立马尔可夫模型。

```
from glob import glob

def genome(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted, # 文件名
                   c.map(open), # 行
                   # 连接所有文件中的行
                   tz.concat,
                   # 去掉每个序列的标题
                   c.filter(is_sequence),
                   # 连接所有行中的字符
                   tz.concat,
                   # 去掉换行符和N
                   c.filter(is_nucleotide))
```

接下来在果蝇基因组上测试一下。

```
# 从ftp://hgdownload.cse.ucsc.edu/goldenPath/dm6/bigZips/下载dm6.fa.gz
# 使用前，用gzip -d dm6.ga.gz命令解压缩文件
dm = 'data/dm6.fa'
model = tz.pipe(dm, genome, c.take(10**7), markov)
# 为了加快速度，使用take，只在前1000万个碱基上运行
# 如果你可以等5~10分钟，就可以省去take这一步
```

查看结果矩阵。

```
print('    ', '      '.join('ACGTacgt'), '\n')
print(model)

     A      C      G      T      a      c      g      t

[[ 0.348  0.182  0.194  0.275  0.     0.     0.     0.   ]
 [ 0.322  0.224  0.198  0.254  0.     0.     0.     0.   ]
 [ 0.262  0.272  0.226  0.239  0.     0.     0.     0.   ]
 [ 0.209  0.199  0.245  0.347  0.     0.     0      0.   ]
 [ 0.003  0.003  0.003  0.003  0.349  0.178  0.166  0.296]
 [ 0.002  0.002  0.003  0.003  0.376  0.195  0.152  0.267]
 [ 0.002  0.003  0.003  0.002  0.281  0.231  0.194  0.282]
 [ 0.002  0.002  0.003  0.003  0.242  0.169  0.227  0.351]]
```

这个结果用图形表示可能会更清楚一些（见图 8-1）。

```
def plot_model(model, labels, figure=None):
    fig = figure or plt.figure()
    ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    im = ax.imshow(model, cmap='magma');
    axcolor = fig.add_axes([0.91, 0.1, 0.02, 0.8])
    plt.colorbar(im, cax=axcolor)
    for axis in [ax.xaxis, ax.yaxis]:
        axis.set_ticks(range(8))
        axis.set_ticks_position('none')
        axis.set_ticklabels(labels)
    return ax

plot_model(model, labels='ACGTacgt');
```

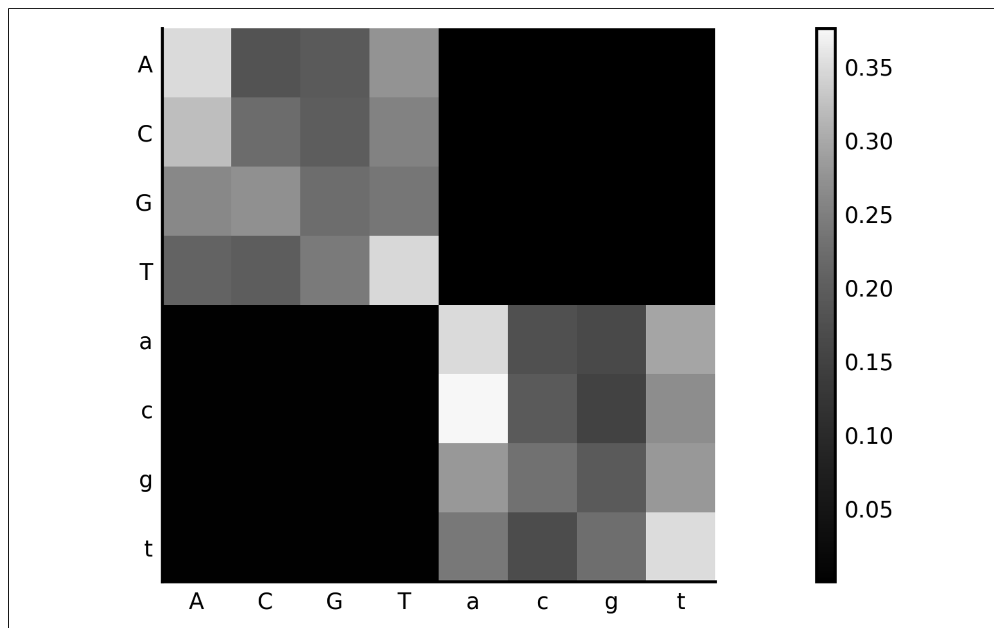


图 8-1：果蝇基因组基因序列的转移概率矩阵

注意 C-A 和 G-C 转移在基因组的重复部分和非重复部分有多大程度的不同，我们可以用这种信息为以前未知的 DNA 序列分类。

练习：在线解压

在管道开始位置添加一个数据解压步骤，这样就无须在硬盘上保留一份解压后的数据了。以果蝇基因组为例，使用 `gzip` 压缩后，所占硬盘空间还不到原来的 1/3。是的，解压也可以通过流实现！



Python 标准库中的 `gzip` 包可以像打开普通文件一样打开 .gz 文件。

学习完本章后，我们希望你至少能够理解这个概念：要想在 Python 中更轻松地进行流操作，可以使用一些抽象方法，就像 Toolz 提供的那样。

流操作可以提高你的工作效率，因为相对于小数据来说，大数据操作需要的时间是线性增加的。在批量分析中，大数据要花费特别长的运行时间，因为操作系统必须保持从内存到硬盘以及反方向的数据传输。如果不使用流操作，一旦出现错误，Python 就会将任务整体拒绝，简单地显示一个 `MemoryError` 错误！在很多分析中，处理更大的数据集并不需要更大的机器。而且，如果程序在小数据上通过测试，那么在大数据上也同样有效！

本章的实际结论是：编写算法或进行分析时，需要先考虑是否可以通过流操作实现。如果可以，那么从一开始就要使用流，未来的你会感激这个决定。否则，以后再想使用流操作，难度会大大增加，产生的严重后果就像图 8-2 中的一样。

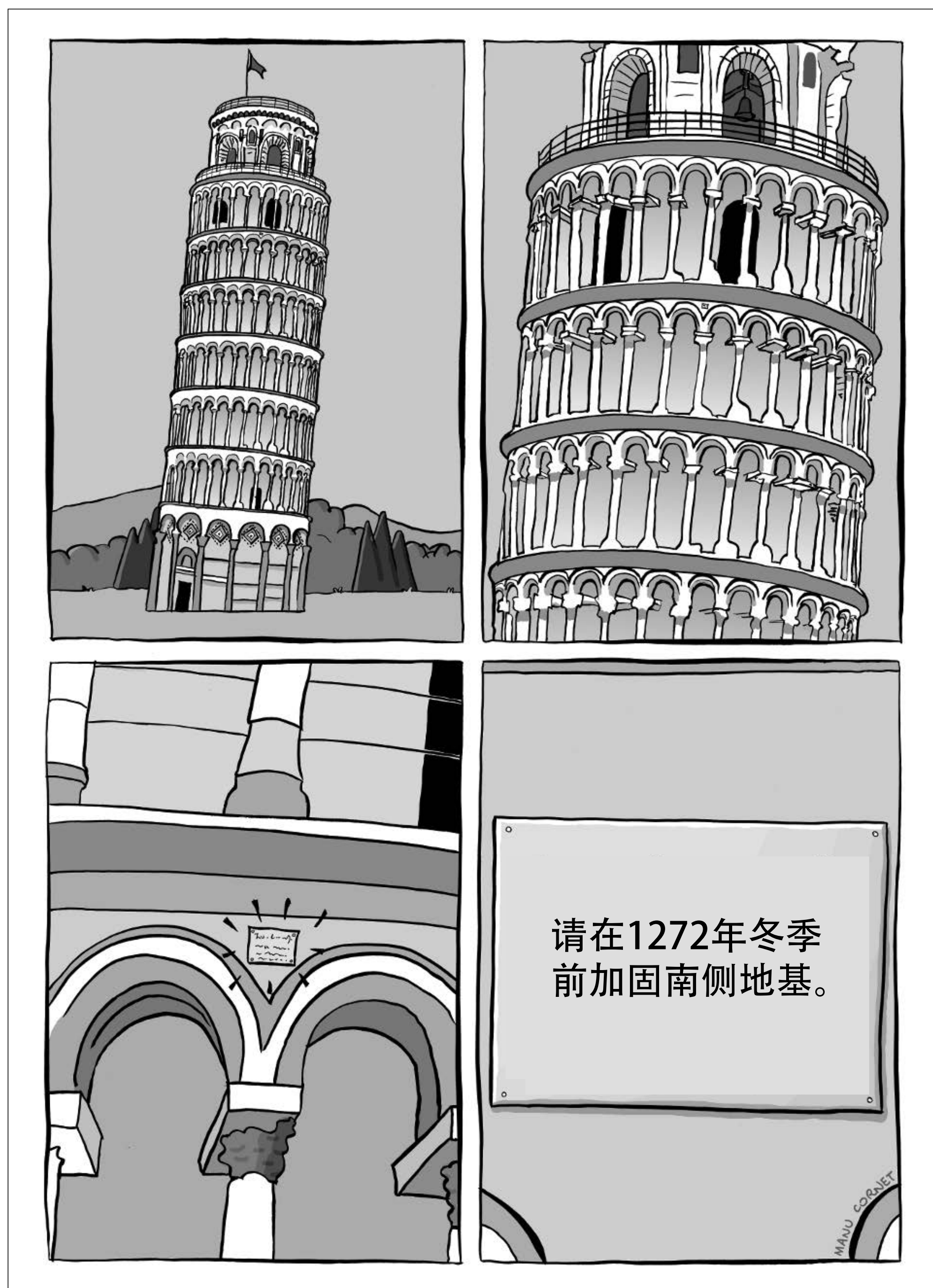


图 8-2：历史上的待办事件（漫画作者 Manu Cornet，经许可使用）

后记

> 品质就是没有人监督也要踏踏实实地做好。
>
> ——亨利·福特

本书的主要目标是促进 NumPy 和 SciPy 的优雅使用。除了教会你如何使用 SciPy 进行有效的科学分析，我们也希望本书能够鼓舞你编写高质量的代码，并意识到高质量的代码是值得追求的。

然后该做什么

如果你已经熟练掌握 SciPy，可以分析遇到的所有数据，那么下一步该做什么呢？本书开头就曾经说过，本书不可能全面介绍 SciPy 库及其所有衍生库。接下来将提供几种对你大有帮助的丰富资源。

邮件列表

前言中提到过，SciPy 是一个大社区。继续学习的一种好方法就是订阅 NumPy、SciPy、pandas、Matplotlib、scikit-image 和你可能感兴趣的其他库的主邮件列表，并定期阅读。

当你在工作中遇到困难时，放心地在邮件列表中寻求帮助吧！列表中都是一些非常友好的家伙！在寻求帮助时，最主要的就是要表明你已经在努力解决问题了，还应该提供一些脚本和足够的样本数据来演示你的问题，并说明你是如何努力解决问题的。

- **不要这样说**："我需要生成一个随机的符合高斯分布的大数组，有人能帮我吗？"
- **不要这样说**："在 https://github.com/ron_obvious 中有一个巨大的库。如果你查看其中的统计库，会发现有一个部分需要随机高斯分布。有人能帮忙看一下吗？"

- **应该这样说**：“我在设法生成一个随机高斯分布大列表，如：`gauss = [np.random.randn()] * 10**5`，但计算 `np.mean(gauss)` 时总是不能像我预想的那样趋近于 0。我哪里做错了吗？以下是完整的脚本。”

GitHub

前言中还提到了 GitHub。我们讨论过的所有代码都是托管在 GitHub 上的：

- NumPy（https://github.com/numpy/numpy）
- SciPy（https://github.com/scipy/scipy）

还有其他代码。当代码没有像你预想的那样工作时，其中可能有 bug。经过一番研究后，你确信自己发现了 bug，那么就应该去相应 GitHub 仓库的“issues”标签页创建一个新问题。这能确保库的开发者获悉该问题，并有望在程序的下一个版本中得到修正。顺便说一下，对于文档中的“bug”，你也可以这样做：如果库的文档有不清楚的地方，那么就提交一个问题！

比提交问题效果更好的做法是**提交 pull request**。提交能改善库文档的 pull request 是你参与开源开发的一个好方法！我们不会详细介绍这个过程，很多图书和资源都可以帮助你。

- Anthony Scopatz 和 Katy huff 合著的《Python 物理学高效计算》一书介绍了 Git 和 GitHub，还有很多其他科学计算话题。
- Peter Bell 和 Brent Beer 合著的《GitHub 入门》一书更加详尽地介绍了 GitHub。
- Software Carpentry 有一门 Git 课程，并且全年在世界各地举办免费的研讨班。
- 部分基于这些课程，一位作者创建了一套关于 Git 和 GitHub pull request 的完整教程，“Open Source Science with Git and GitHub”（http://jni.github.io/git-tutorial/）。
- 最后，GitHub 上的很多开源项目都有一个 CONTRIBUTING 文件（https://github.com/scikit-image/scikit-image/blob/master/CONTRIBUTING.txt），其中包含了为项目贡献代码的指南。

因此，你可以就 GitHub 得到很多帮助！

我们鼓励你尽可能为 SciPy 生态系统做贡献，这不仅可以使库更好地为他人服务，也是提高你编程能力的最佳方法之一。每提交一个 pull request，你都会收到关于自己代码的反馈，可以帮助你改进代码。你还能更加熟悉 GitHub 贡献的过程和规则，这在当今职场是非常重要的技能。

会议

基于同样的理由，我们强烈建议你参加该领域的一个编程会议。每年都在奥斯汀举办的 SciPy 会议非常精彩，如果你喜欢本书，那么它就是你的最佳选择。欧洲也有一个同样的会议，叫作 EuroSciPy，每两年更换一次主办城市。最后，最著名的 PyCon 会议是在美国举行的，但世界各地都有分支会议，比如澳大利亚的 PyCon-AU，它在主会议的前一天会有一个主题为“科学与数据”的小型会议。

不管选择哪个会议，你都应该参加一下会议末尾的**编程冲刺活动**。编程冲刺是一种组队编程的高强度训练，不管你的编程技能如何，它都是学习如何为开源软件做贡献的极好机会。本书作者之一（胡安）就是从此走上开源之路的。

SciPy之外

SciPy 库不只是用 Python 编写的，它还使用了高度优化的 C 和 Fortran 代码，它们可以通过接口与 Python 连接。SciPy 与 NumPy 及其他相关库一起，可以解决科学数据分析中的大多数问题，并提供用以解决这些问题的高效函数。但有时某个科学问题会与 SciPy 中的现有功能不太匹配，而纯 Python 解决方案又速度太慢，并不实用，该怎么办呢？

Micha Gorelick 和 Ian Ozsvald 合著的《Python 高性能编程》一书介绍了这种情况下需要掌握的知识：如何找到确实需要高性能的地方，以及实现高性能的几种方法。强烈推荐你阅读一下此书。

接下来再简单介绍两种与 SciPy 关系特别密切的编程语言。

首先是 Cython，它是 Python 的一个变种，可以将 Python 脚本编译成 C 代码，然后再导入 Python。为 Python 变量提供一些类型注解，意味着编译后的 C 代码可以比相应的 Python 代码快几百甚至几千倍。Cython 现在已经是行业标准，大量应用于 NumPy、SciPy 和很多相关库（如 scikit-image），以在基于数组的代码中提供快速算法。Kurt Smith 撰写了 *Cython* 一书来介绍这门语言的基础知识。

比 Cython 更易于使用的是 Numba，它是一种实时（JIT，just-in-time）编译器，用来编译基于数组的 Python 脚本。JIT 可以在函数运行时推断出函数所有参数和输出的类型，并将代码编译成适合这些类型的高效格式。在 Numba 代码中，无须注解类型：Numba 会在第一次调用函数时推断出这些类型。但是，你必须确保只使用基本数据类型（整型、浮点型等）和数组，不使用更复杂的 Python 对象。在这种情况下，Numba 可以将 Python 代码向下编译成非常高效的代码，计算速度可以提高好几个数量级。

Numba 还很年轻，但已体现出实用性。更重要的是，它展示出了 Python JIT 的可能性，Python JIT 已经变得非常普遍：Python 3.6 新增了功能，以便更容易地使用各种新 JIT（Pyjion JIT 就基于这些新增功能）。在胡安的博客上，你可以看到 Numba 应用的一些示例，以及将它与 SciPy 协同使用的方法。当然，Numba 自己也有一个非常活跃和友好的邮件列表。

为本书做贡献

本书的源文件托管在 GitHub（以及本书网站）上。与你为任何其他开源项目做贡献一样，如果你发现了本书的技术错误或拼写错误，可以提交问题或 pull request，我们将非常感激。

为了演示 SciPy 和 NumPy 库的各种功能，我们使用了一些能找到的最佳代码。如果你有更好的示例，请在仓库中提交问题，我们很乐于在未来的版本中使用这些示例。

本书的 Twitter 账号是 @elegantscipy，如果想要讨论本书，就给我们发消息吧。作者的账号分别是 @jnuneziglesias、@stefanvdwalt 和 @stefanvdwalt。

如果你利用本书的思想或代码在科学研究中取得了进展，请一定告诉我们，我们特别希望听到这样的消息，这正是 SciPy 的价值所在！

后会有期

希望你喜欢本书并觉得它很有用。如果确实如此，请转告你的朋友，并在邮件列表、会议、GitHub 和 Twitter 上积极互动。感谢你阅读本书，希望你能从本书中收获很多！

附录

练习答案

A.1 答案：为图像覆盖一个网格

这是“练习：为图像覆盖一个网格”的答案。

可以用 NumPy 的切片操作来选择网格的行，将它们设置为蓝色，然后再选择列，将它们也设置为蓝色（见图 A-1）。

```
def overlay_grid(image, spacing=128):
    """Return an image with a grid overlay, using the provided spacing.

    Parameters
    ----------
    image : array, shape (M, N, 3)
        The input image.
    spacing : int
        The spacing between the grid lines.

    Returns
    -------
    image_gridded : array, shape (M, N, 3)
        The original image with a blue grid superimposed.
    """
    image_gridded = image.copy()
    image_gridded[spacing:-1:spacing, :] = [0, 0, 255]
    image_gridded[:, spacing:-1:spacing] = [0, 0, 255]
    return image_gridded

plt.imshow(overlay_grid(astro, 128));
```

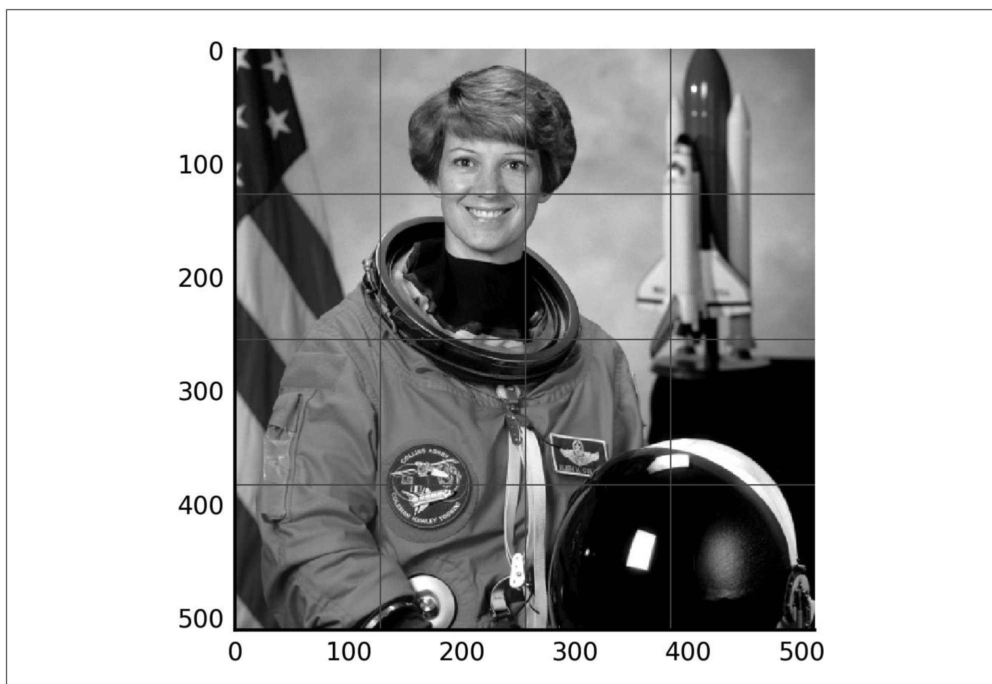


图 A-1：覆盖了网格的宇航员图片

注意，我们用 `-1` 表示轴上的最后一个值，就像 Python 索引一样。你可以省略这个值，但含义会有些不同。如果没有这个值（即 `spacing::spacing`），那么到达数组末尾时会包含最后的行或列。如果将 `-1` 作为结束索引，最后一行就不能被选择。在覆盖网格的情况下，或许这才是我们要做的。

A.2　答案：康威的生命游戏

这是“练习：康威的生命游戏”的答案。

Nicolas Rougier（@NPRougier）在其 100 NumPy Exercises 的第 79 个练习中提供了一个纯 NumPy 解决方案。

```
def next_generation(Z):
    N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
         Z[1:-1,0:-2]                + Z[1:-1,2:] +
         Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])

    # 应用规则
    birth = (N==3) & (Z[1:-1,1:-1]==0)
    survive = ((N==2) | (N==3)) & (Z[1:-1,1:-1]==1)
    Z[...] = 0
    Z[1:-1,1:-1][birth | survive] = 1
    return Z
```

然后可以用以下代码开始一个新的游戏板。

```
random_board = np.random.randint(0, 2, size=(50, 50))
n_generations = 100
for generation in range(n_generations):
    random_board = next_generation(random_board)
```

使用通用滤波器会更容易。

```
def nextgen_filter(values):
    center = values[len(values) // 2]
    neighbors_count = np.sum(values) - center
    if neighbors_count == 3 or (center and neighbors_count == 2):
        return 1.
    else:
        return 0.

def next_generation(board):
    return ndi.generic_filter(board, nextgen_filter,
                              size=3, mode='constant')
```

这样做的好处是，有些生命游戏的演化使用了一种称为**环形板**的规则。也就是说，游戏板的左端和右端可以“环绕”并连接在一起，上端和底端也是如此。`generic_filer` 使得修改解决方案来实现这种环形板简直易如反掌。

```
def next_generation_toroidal(board):
    return ndi.generic_filter(board, nextgen_filter,
                              size=3, mode='wrap')
```

现在可以模拟这种环形板在几个时代的变化了。

```
random_board = np.random.randint(0, 2, size=(50, 50))
n_generations = 100
for generation in range(n_generations):
    random_board = next_generation_toroidal(random_board)
```

A.3　答案：Sobel梯度幅值

这是“练习：Sobel 梯度幅值”的答案。

```
hsobel = np.array([[ 1,  2,  1],
                   [ 0,  0,  0],
                   [-1, -2, -1]])

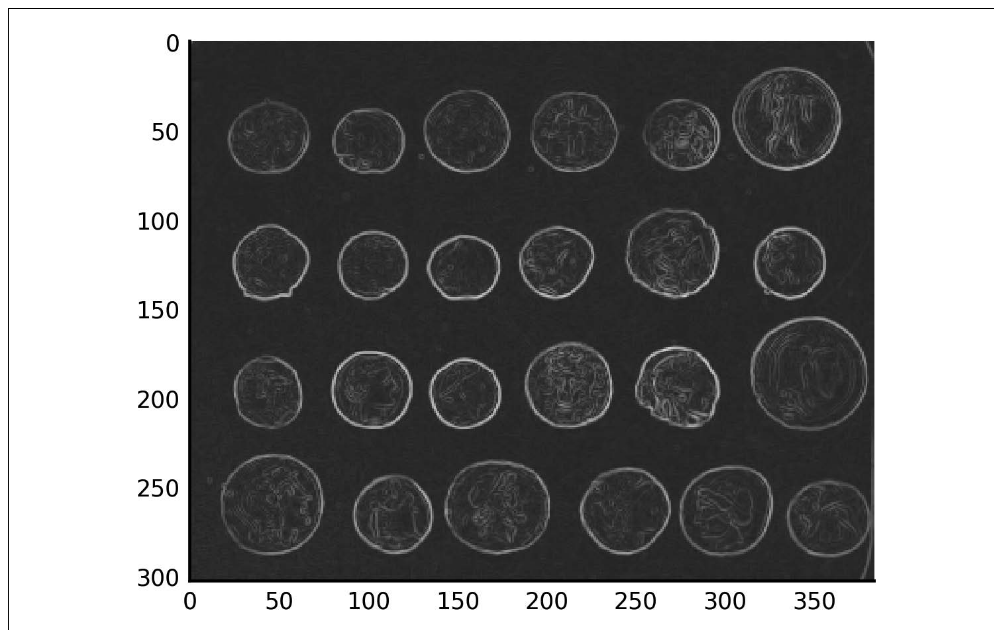
vsobel = hsobel.T

hsobel_r = np.ravel(hsobel)
vsobel_r = np.ravel(vsobel)

def sobel_magnitude_filter(values):
    h_edge = values @ hsobel_r
    v_edge = values @ vsobel_r
    return np.hypot(h_edge, v_edge)
```

现在我们在钱币图案上试验一下。

```
sobel_mag = ndi.generic_filter(coins, sobel_magnitude_filter, size=3)
plt.imshow(sobel_mag, cmap='viridis');
```



A.4　答案：用SciPy进行曲线拟合

这是“练习：用 SciPy 进行曲线拟合”的答案。

来看一下 `curve_fit` 的文档字符串的开头。

```
Use nonlinear least squares to fit a function, f, to data.

Assumes ``ydata = f(xdata, *params) + eps``

Parameters
----------
f : callable
    The model function, f(x, ...).  It must take the independent
    variable as the first argument and the parameters to fit as
    separate remaining arguments.
xdata : An M-length sequence or an (k,M)-shaped array
    for functions with k predictors.
    The independent variable where the data is measured.
ydata : M-length sequence
    The dependent data --- nominally f(xdata, ...)
```

看起来我们只需要提供一个能接受数据点和某些参数的函数，该函数返回预测值。在示例中，我们需要累积剩余频率 $f(d)$ 与 d^{-y} 成正比，也就是说，我们需要 $f(d) = \alpha d^{-\text{gamma}}$。

```
def fraction_higher(degree, alpha, gamma):
    return alpha * degree ** (-gamma)
```

当 d>10 时，我们需要待拟合的 X 和 Y 数据。

```
x = 1 + np.arange(len(survival))
valid = x > 10
x = x[valid]
y = survival[valid]
```

现在可以通过 `curve_fit` 得到拟合参数。

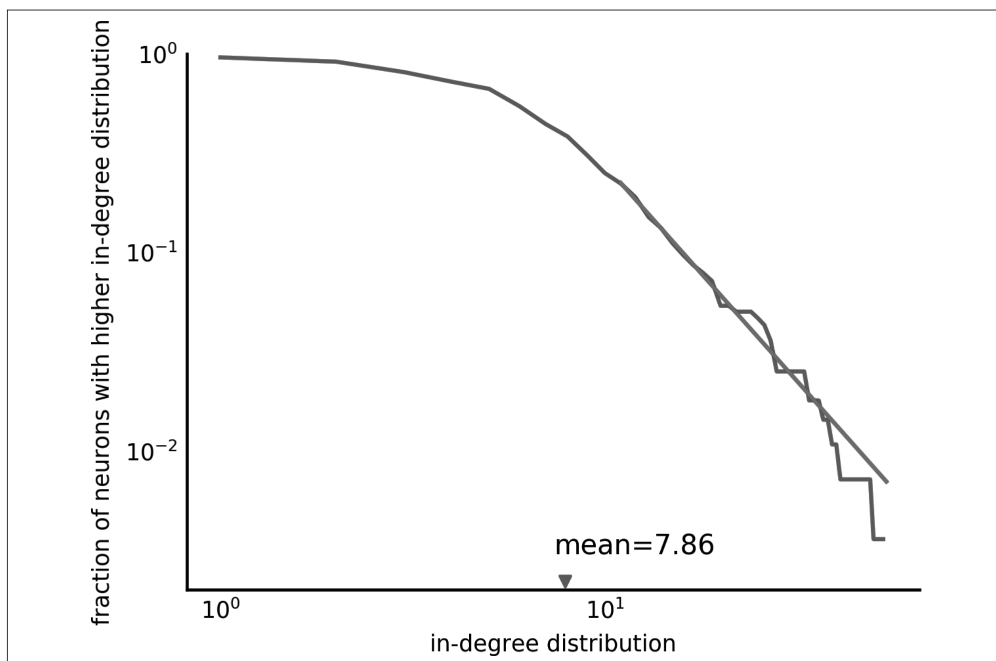
```
from scipy.optimize import curve_fit

alpha_fit, gamma_fit = curve_fit(fraction_higher, x, y)[0]
```

绘制结果，看看效果如何。

```
y_fit = fraction_higher(x, alpha_fit, gamma_fit)

fig, ax = plt.subplots()
ax.loglog(np.arange(1, len(survival) + 1), survival)
ax.set_xlabel('in-degree distribution')
ax.set_ylabel('fraction of neurons with higher in-degree distribution')
ax.scatter(avg_in_degree, 0.0022, marker='v')
ax.text(avg_in_degree - 0.5, 0.003, 'mean=%.2f' % avg_in_degree)
ax.set_ylim(0.002, 1.0)
ax.loglog(x, y_fit, c='red');
```



瞧！一张完整的 6B 图形，完美的拟合！

A.5　答案：图像卷积

这是“练习：图像卷积”的答案。

```
from scipy import signal

x = np.random.random((50, 50))
y = np.ones((5, 5))

L = x.shape[0] + y.shape[0] - 1
Px = L - x.shape[0]
Py = L - y.shape[0]

xx = np.pad(x, ((0, Px), (0, Px)), mode='constant')
yy = np.pad(y, ((0, Py), (0, Py)), mode='constant')

zz = np.fft.ifft2(np.fft.fft2(xx) * np.fft.fft2(yy)).real
print('Resulting shape:', zz.shape, ' <-- Why?')

z = signal.convolve2d(x, y)
print('Results are equal?', np.allclose(zz, z))

Resulting shape: (54, 54)  <-- Why?
Results are equal? True
```

A.6　答案：混淆矩阵的计算复杂度

这是“练习：混淆矩阵的计算复杂度”的答案。

你应该记得在第 1 章中，`arr == k` 可以创建一个与 `arr` 相同大小的布尔值（`True` 或 `False`）数组。正如你所料，这需要对 `arr` 进行完整的遍历。因此，在上面的解决方案中，对于 `pred` 和 `gt` 中的每一个值组合，我们都要完整遍历 `pred` 和 `gt`。原则上来说，只需要遍历这两个数组一次，就能算出 `cont`，因此这种多次遍历是没有效率的。

A.7　答案：计算混淆矩阵的另一种方法

这是“练习：计算混淆矩阵的另一种方法”的答案。

这里只提供两种方法，但其实还有很多其他方法。

第一种方法用 Python 内置的 `zip` 函数将 `pred` 和 `gt` 中的标签成对组合起来。

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    for i, j in zip(pred, gt):
        cont[i, j] += 1
    return cont
```

第二种方法是在 `pred` 和 `gt` 的所有可能标号之间迭代，并手工取出每个数组中相应的值。

```
def confusion_matrix1(pred, gt):
    cont = np.zeros((2, 2))
    for idx in range(len(pred)):
        i = pred[idx]
        j = gt[idx]
        cont[i, j] += 1
    return cont
```

在这两种方法中，第一种更加“Python 化”，第二种更容易转换和编译成 C、Cython 和 Numba 这样的语言（这是另外一本书的主题），从而提高运行速度。

A.8 答案：计算混淆矩阵

这是“练习：多类混淆矩阵”的答案。

我们只需要对两个输入数组做一次初始遍历，以确定最大的标号。然后对最大标号加 1，以解决 0 标号和 Python 索引从 0 开始的问题。接着就可以创建矩阵，并用和前一个示例相同的代码填充它。

```
def general_confusion_matrix(pred, gt):
    n_classes = max(np.max(pred), np.max(gt)) + 1
    cont = np.zeros((n_classes, n_classes))
    for i, j in zip(pred, gt):
        cont[i, j] += 1
    return cont
```

A.9 答案：COO表示

这是“练习：COO 表示”的答案。

先列出数组中的非零元素，从左到右，从上到下，就像英文阅读顺序一样。

```
s2_data = np.array([6, 1, 2, 4, 5, 1, 9, 6, 7])
```

然后按同样顺序列出这些值的行索引。

```
s2_row = np.array([0, 1, 1, 1, 1, 2, 3, 4, 4])
```

最后是列索引。

```
s2_col = np.array([2, 0, 1, 3, 4, 1, 0, 3, 4])
```

通过检查行和列两个方向上是否相等，可以知道这样能否生成正确的矩阵。

```
s2_coo0 = sparse.coo_matrix(s2)
print(s2_coo0.data)
print(s2_coo0.row)
print(s2_coo0.col)

[6 1 2 4 5 1 9 6 7]
[0 1 1 1 1 2 3 4 4]
[2 0 1 3 4 1 0 3 4]
```

并且：

```
s2_coo1 = sparse.coo_matrix((s2_data, (s2_row, s2_col)))
print(s2_coo1.toarray())

[[0 0 6 0 0]
 [1 2 0 4 5]
 [0 1 0 0 0]
 [9 0 0 0 0]
 [0 0 0 6 7]]
```

A.10 答案：图像旋转

这是“练习：图像旋转”的答案。

可以通过矩阵乘法实现**组合**转换。我们知道如何围绕原点旋转图像，也知道如何滑动图像。因此，我们要做的就是先滑动图像，使其中心位于原点，然后旋转图像，再滑动回原来的位置。

```
def transform_rotate_about_center(shape, degrees):
    """Return the homography matrix for a rotation about an image center."""
    c = np.cos(np.deg2rad(angle))
    s = np.sin(np.deg2rad(angle))

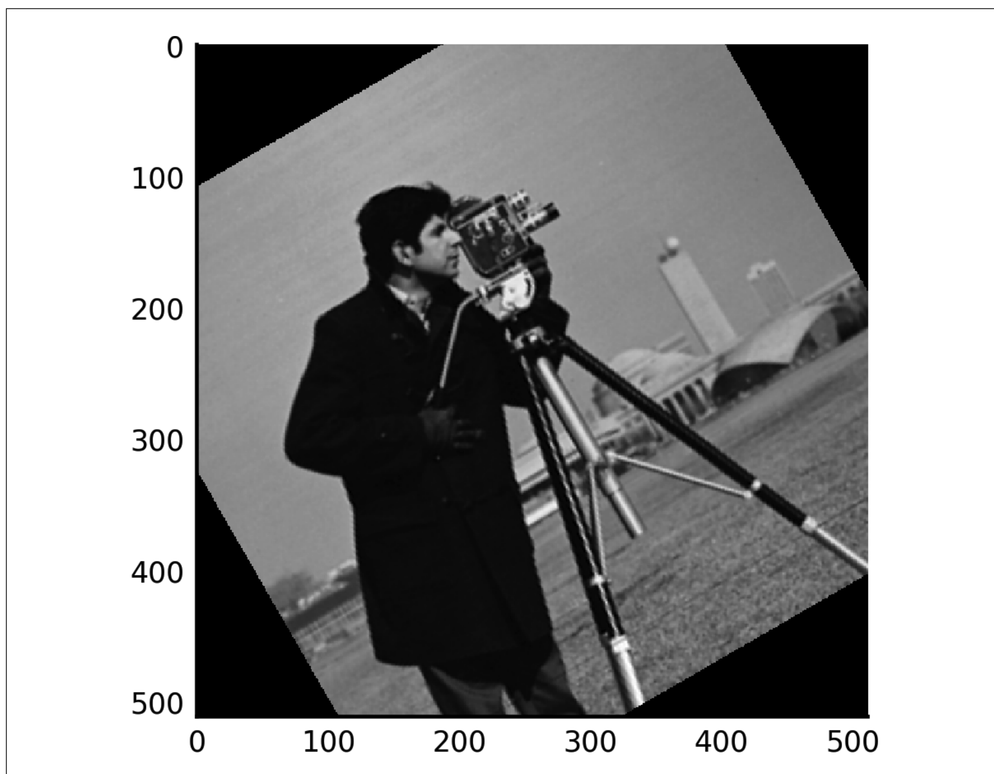
    H_rot = np.array([[c, -s,  0],
                      [s,  c,  0],
                      [0,  0,  1]])
    # 计算图像中心坐标
    center = np.array(image.shape) / 2
    # 将图像中心移到原点的矩阵
    H_tr0 = np.array([[1, 0, -center[0]],
                      [0, 1, -center[1]],
                      [0, 0,          1]])
    # 将图像中心移回原来位置的矩阵
    H_tr1 = np.array([[1, 0, center[0]],
                      [0, 1, center[1]],
                      [0, 0,         1]])
    # 完整的转换矩阵
    H_rot_cent = H_tr1 @ H_rot @ H_tr0

    sparse_op = homography(H_rot_cent, image.shape)

    return sparse_op
```

我们可以测试一下效果。

```
tf = transform_rotate_about_center(image.shape, 30)
plt.imshow(apply_transform(image, tf));
```



A.11　答案：减少内存占用

这是“练习：减少内存占用”的答案。

我们使用 np.ones 建立的数组是只读的，它只能用作被 coo_matrix 加总的值。可以用 broadcast_to 创建一个类似的数组，其中只有一个元素，“虚拟”地重复 n 次。

```
def confusion_matrix(pred, gt):
    n = pred.size
    ones = np.broadcast_to(1., n)  # 虚拟数组，1个元素重复n次
    cont = sparse.coo_matrix((ones, (pred, gt)))
    return cont
```

确认这段代码像所预想的那样奏效。

```
cont = confusion_matrix(pred, gt)
print(cont.toarray())

[[ 3. 1.]
 [ 2. 4.]]
```

我们没有建立和原始数据一样大的数组，只建立了一个大小为 1 的数组。随着处理的数据集越来越大，这种优化方法会变得越来越重要。

A.12 答案：计算条件熵

这是“练习：计算条件熵”的答案。

要得到联合概率表，只需要将表格除以它的总和，这个例子中就是 12。

```
print('table total:', np.sum(p_rain_g_month))
p_rain_month = p_rain_g_month / np.sum(p_rain_g_month)

table total: 12.0
```

这样就可以计算出给定 `rain` 时 `month` 的条件熵了。（同理，如果我们知道正在下雨，那么一般还需要知道多少其他信息才能确定现在是哪个月份？）

```
p_rain = np.sum(p_rain_month, axis=0)
p_month_g_rain = p_rain_month / p_rain
Hmr = np.sum(p_rain * p_month_g_rain * np.log2(1 / p_month_g_rain))
print(Hmr)

3.5613602411
```

与月份本身的熵比较一下。

```
p_month = np.sum(p_rain_month, axis=1) # 11/12，但这种方法更有普遍性
Hm = np.sum(p_month * np.log2(1 / p_month))
print(Hm)

3.58496250072
```

可见，知道今天是否下雨可以使得猜测现在是哪个月份的准确度提高 2 个百分点！但千万不要根据这种猜测种庄稼。

A.13 答案：旋转矩阵

这是“练习：旋转矩阵”的答案。

第1部分

```
import numpy as np

theta = np.deg2rad(45)
R = np.array([[np.cos(theta), -np.sin(theta), 0],
              [np.sin(theta),  np.cos(theta), 0],
              [            0,              0, 1]])

print("R times the x-axis:", R @ [1, 0, 0])
print("R times the y-axis:", R @ [0, 1, 0])
print("R times a 45 degree vector:", R @ [1, 1, 0])

R times the x-axis: [ 0.70710678  0.70710678   0.        ]
R times the y-axis: [-0.70710678  0.70710678   0.        ]
R times a 45 degree vector: [  1.11022302e-16    1.41421356e+00  0.00000000e+00]
```

第2部分

因为用 $\boldsymbol{R}$ 乘以一个向量会将它旋转 45 度，所以再乘以一次 $\boldsymbol{R}$ 就会将向量旋转 90 度。矩阵乘法满足结合律，即 $\boldsymbol{R}(\boldsymbol{Rv}) = (\boldsymbol{RR})\boldsymbol{v}$，因此 $\boldsymbol{S} = \boldsymbol{RR}$ 可以将向量沿 z 轴旋转 90 度。

```
S = R @ R
S @ [1, 0, 0]

array([  2.22044605e-16,   1.00000000e+00,   0.00000000e+00])
```

第3部分

```
print("R @ z-axis:", R @ [0, 0, 1])

R @ z-axis: [ 0.  0.  1.]
```

$\boldsymbol{R}$ 同时旋转 x 轴和 y 轴，不旋转 z 轴。

第4部分

查看 `eig` 的文档，可以知道它返回两个值：一个一维的特征值数组，还有一个二维的数组，其中每列包含与相应特征值对应的特征向量。

```
np.linalg.eig(R)

(array([ 0.70710678+0.70710678j, 0.70710678-0.70710678j, 1.00000000+0.j      ]),
 array([[ 0.70710678+0.j        , 0.70710678-0.j        , 0.00000000+0.j      ],
        [ 0.00000000-0.70710678j, 0.00000000+0.70710678j, 0.00000000+0.j      ],
        [ 0.00000000-0.j        , 0.00000000+0.j        , 1.00000000+0.j      ]]))
```

除了一些复数特征值和特征向量，还可以看到特征值 1 对应的特征向量是 $[0, 0, 1]^T$。

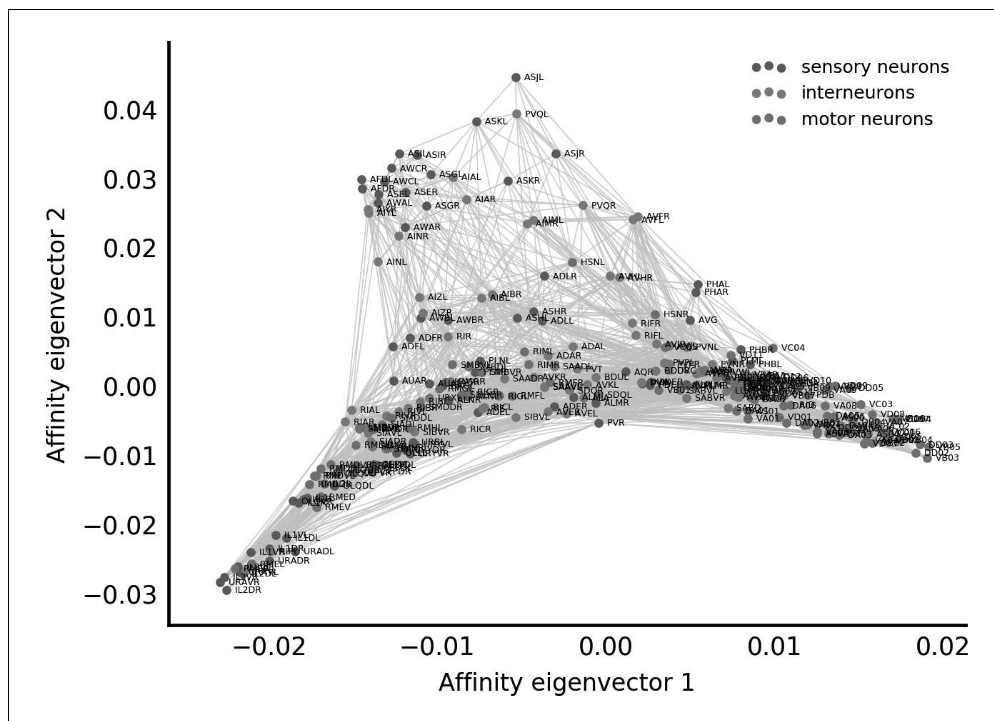
A.14　答案：显示近邻视图

这是“练习：显示近邻视图”的答案。

在近邻视图中，我们不在 y 轴上使用处理深度，而是像 x 轴一样，使用矩阵 $\boldsymbol{Q}$ 标准化的第三小的特征向量。（与对 x 轴的处理一样，如果需要，则取相反数！）

```
y = Dinv2 @ Vec[:, 2]
asjl_index = np.argwhere(neuron_ids == 'ASJL')
if y[asjl_index] < 0:
    y = -y

plot_connectome(x, y, C, labels=neuron_ids, types=neuron_types,
                type_names=['sensory neurons', 'interneurons',
                            'motor neurons'],
                xlabel='Affinity eigenvector 1',
                ylabel='Affinity eigenvector 2')
```



A.15 接受挑战：稀疏矩阵线性代数

这是“练习挑战：稀疏矩阵线性代数”的答案。

为了解决这个挑战，我们将使用一个小网络，因为这样更容易实现可视化。这个挑战的后续部分将用这种技术来分析更大的网络。

首先从邻接矩阵 A 开始，使用稀疏矩阵的格式。这个示例使用 CSR 格式，它是应用线性代数最常见的格式。我们在所有矩阵名称后面都加上一个 s，表示它们是稀疏格式的。

```
from scipy import sparse
import scipy.sparse.linalg

As = sparse.csr_matrix(A)
```

用同样的方式建立连接矩阵。

```
Cs = (As + As.T) / 2
```

为了得到度矩阵，可以使用“对角”稀疏矩阵格式，它可以保存对角阵和非对角阵。

```
degrees = np.ravel(Cs.sum(axis=0))
Ds = sparse.diags(degrees)
```

轻松得到拉普拉斯矩阵。

```
Ls = Ds - Cs
```

现在要得出处理深度。注意，得出拉普拉斯矩阵的伪逆是不可能的，因为它会是一个稠密矩阵（稀疏矩阵的逆通常不是稀疏矩阵）。但是，我们要使用伪逆计算出一个向量 z，以满足 $\boldsymbol{L}\boldsymbol{z} = \boldsymbol{b}$，其中 $\boldsymbol{b} = \boldsymbol{C} \odot \text{sign}(\boldsymbol{A} - \boldsymbol{A}^{\text{T}})\ \boldsymbol{1}$（参见 Varshney 等人的补充资料）。对于稠密矩阵，可以简单地使用 $z = L^{+}b$。但对于稀疏矩阵，可以使用 `sparse.linalg.isolve` 中的一个**求解器**（参见 6.3.2 节中的“求解程序”部分），在已知 $\boldsymbol{L}$ 和 $\boldsymbol{b}$ 时得到 z 向量，而不用求逆！

```
b = Cs.multiply((As - As.T).sign()).sum(axis=1)
z, error = sparse.linalg.isolve.cg(Ls, b, maxiter=10000)
```

最后，必须找出度标准化拉普拉斯矩阵 $\boldsymbol{Q}$ 对应于其第二小和第三小的特征值的特征向量。

第 5 章中介绍过，稀疏矩阵的数值型数据在 `.data` 属性中。我们用这个属性求出度矩阵的倒数。

```
Dsinv2 = Ds.copy()
Dsinv2.data = 1 / np.sqrt(Ds.data)
```

最后用 SciPy 的稀疏线性代数函数找出所需的特征向量。矩阵 $\boldsymbol{Q}$ 是对称的，因此可以用适用于对称矩阵的 `eigsh` 函数来计算特征向量。我们用 `which` 关键字参数来指定需要与最小的特征值对应的特征向量，并用 `k` 指定需要 3 个最小的特征值。

```
Qs = Dsinv2 @ Ls @ Dsinv2
vals, Vecs = sparse.linalg.eigsh(Qs, k=3, which='SM')
sorted_indices = np.argsort(vals)
Vecs = Vecs[:, sorted_indices]
```

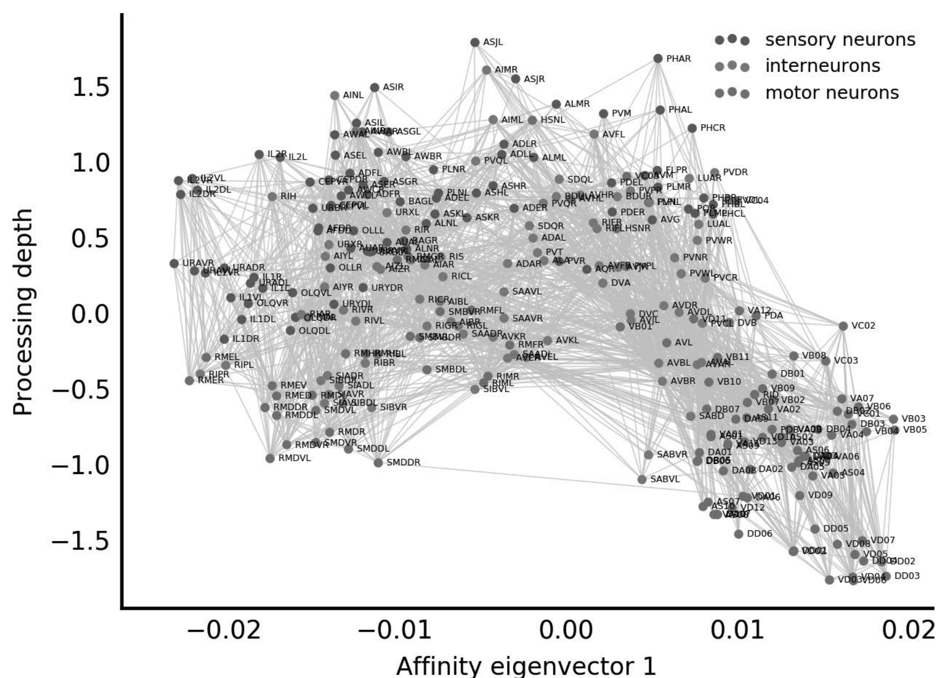
最后，对特征向量进行标准化，以得到 x 轴和 y 轴的坐标（如果需要，则取相反数）。

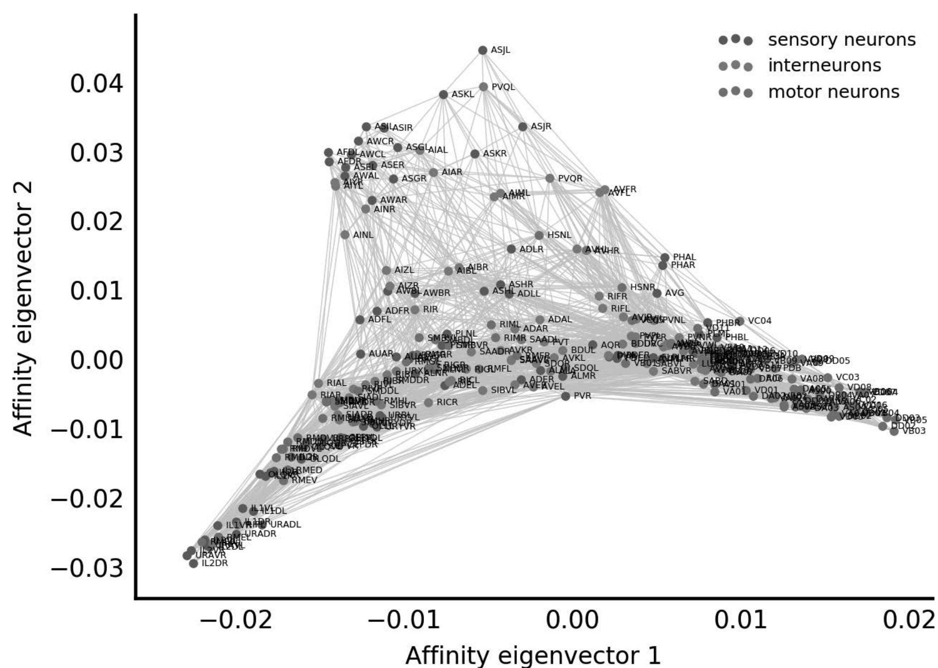
```
_dsinv, x, y = (Dsinv2 @ Vecs).T
if x[vc2_index] < 0:
    x = -x
if y[asjl_index] < 0:
    y = -y
```

（注意，与最小特征值对应的特征向量总是一个全部为 1 的向量，我们对它没有兴趣。）然后就可以绘制出下面的图形了！

```
plot_connectome(x, z, C, labels=neuron_ids, types=neuron_types,
                type_names=['sensory neurons', 'interneurons',
                            'motor neurons'],
                xlabel='Affinity eigenvector 1', ylabel='Processing depth')

plot_connectome(x, y, C, labels=neuron_ids, types=neuron_types,
                type_names=['sensory neurons', 'interneurons',
                            'motor neurons'],
                xlabel='Affinity eigenvector 1',
                ylabel='Affinity eigenvector 2')
```


sensory neurons
interneurons
motor neurons
Processing depth
Affinity eigenvector 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−0.02
−0.01
0.00
0.01
0.02


sensory neurons
interneurons
motor neurons
Affinity eigenvector 2
Affinity eigenvector 1
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.02
−0.01
0.00
0.01
0.02

A.16 答案：处理悬挂节点

这是“练习：处理悬挂节点”的答案。

为了得到随机矩阵，转移矩阵所有列的总和必须为 1。当一个物种不是其他物种的食物时，就不满足这个条件，因为这一列全是 0。然而，如果将这一列都换成 $1/n$，那代价就太大了。

关键是要知道，对于转移矩阵和当前概率向量的相乘，矩阵中的**所有行**贡献**相同的量**。也就是说，这些列加在一起会为这次迭代中的相乘结果增加一个单一值。是什么值呢？就是 $1/n$ 乘以 $\boldsymbol{r}$ 中对应悬挂节点的那个元素。这可以表示为两个向量的点积，一个向量在对应悬挂节点的位置上是 $1/n$，其他位置都是 0，另一个向量是当前迭代中的向量 $\boldsymbol{r}$。

```
def power2(Trans, damping=0.85, max_iter=10**5):
    n = Trans.shape[0]
    dangling = (1/n) * np.ravel(Trans.sum(axis=0) == 0)
    r0 = np.full(n, 1/n)
    r = r0
    for _ in range(max_iter):
        rnext = (damping * (Trans @ r + dangling @ r) +
                 (1 - damping) / n)
        if np.allclose(rnext, r):
        return rnext
    else:
        r = rnext
return r
```

手动迭代几次试一下。注意，如果开始时用的是一个随机向量（所有元素相加等于 1），那么下一次迭代得到的还是随机向量。因此，从这个函数输出的 PageRank 是一个真正的概率向量，向量值表示沿着食物链中的链接最终得到某个物种的概率。

A.17 答案：验证方法

这是“练习：不同特征向量方法的等价性”的答案。

`np.corrcoef` 可以给出一个向量列表中的所有向量对之间的皮尔逊相关系数。这个系数等于 1，当且仅当两个向量彼此之间是标量倍数关系。因此，相关系数为 1 就足以表明以上方法能生成同样的排名。

```
pagerank_power = power(Trans)
pagerank_power2 = power2(Trans)
np.corrcoef([pagerank, pagerank_power, pagerank_power2])

array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

A.18 答案：修改对齐函数

这是“练习：修改对齐函数”的答案。

我们在金字塔的较高级别使用 basin hopping 方法，较低级别则使用 Powell 方法，因为 basin hopping 在全分辨率图像上运行的计算开销太大了。

```
def align(reference, target, cost=cost_mse, nlevels=7, method='Powell'):
    pyramid_ref = gaussian_pyramid(reference, levels=nlevels)
    pyramid_tgt = gaussian_pyramid(target, levels=nlevels)

    levels = range(nlevels, 0, -1)
    image_pairs = zip(pyramid_ref, pyramid_tgt)

    p = np.zeros(3)

    for n, (ref, tgt) in zip(levels, image_pairs):
        p[1:] *= 2
        if method.upper() == 'BH':
            res = optimize.basinhopping(cost, p,
                          minimizer_kwargs={'args': (ref, tgt)})
       if n <= 4: # 避免在低级别上使用basin hopping方法
           method = 'Powell'
    else:
        res = optimize.minimize(cost, p, args=(ref, tgt), method='Powell')
    p = res.x
    # 输出当前级别，每次都覆盖上一次的输出（就像进度条一样）
    print(f'Level: {n}, Angle: {np.rad2deg(res.x[0]) :.3}, '
          f'Offset: ({res.x[1] * 2**n :.3}, {res.x[2] * 2**n :.3}), '
          f'Cost: {res.fun :.3}', end='\r')

print('') # 对齐完成后开始新一行
return make_rigid_transform(p)
```

现在试验一下对齐函数。

```
from skimage import util

theta = 50
rotated = transform.rotate(astronaut, theta)
rotated = util.random_noise(rotated, mode='gaussian',
                            seed=0, mean=0, var=1e-3)

tf = align(astronaut, rotated, nlevels=8, method='BH')
corrected = transform.warp(rotated, tf, order=3)

f, (ax0, ax1, ax2) = plt.subplots(1, 3)
ax0.imshow(astronaut)
ax0.set_title('Original')
ax1.imshow(rotated)
ax1.set_title('Rotated')
ax2.imshow(corrected)
ax2.set_title('Registered')
for ax in (ax0, ax1, ax2):
    ax.axis('off')

Level: 1, Angle: -50.0, Offset: (-2.09e+02, 5.74e+02), Cost: 0.0385
```



成功了！ basin hopping 能够恢复正确的对齐，即使在 `minimize` 函数遇到困难时也是如此。

A.19 答案：scikit-learn库

这是“练习：流数据的主成分分析”的答案。

首先写一个函数来训练模型。这个函数应该接受一个样本流，并输出一个主成分分析模型。这个模型可以将新样本从初始的 N 维空间投射到主成分空间，从而实现新样本的**转换**。

```
import toolz as tz
from toolz import curried as c
from sklearn import decomposition
from sklearn import datasets
import numpy as np

def streaming_pca(samples, n_components=2, batch_size=100):
    ipca = decomposition.IncrementalPCA(n_components=n_components,
                                        batch_size=batch_size)
    tz.pipe(samples,  # 一维数组迭代器
            c.partition(batch_size), # 元组的迭代器
            c.map(np.array), # 二维数组迭代器
            c.map(ipca.partial_fit), # 对样本中的每个元素执行partial_fit
            tz.last) # 通过管道吸入数据流
    return ipca
```

现在可以用这个函数**训练**（或**拟合**）一个主成分分析模型。

```
reshape = tz.curry(np.reshape)

def array_from_txt(line, sep=',', dtype=np.float):
    return np.array(line.rstrip().split(sep), dtype=dtype)

with open('data/iris.csv') as fin:
    pca_obj = tz.pipe(fin, c.map(array_from_txt), streaming_pca)
```

最后，可以通过模型的 `transform` 函数对原始样本进行流式处理，并将处理结果汇集成一个行和列分别为 `n_samples` 和 `n_components` 的数据矩阵。

```
with open('data/iris.csv') as fin:
    components = tz.pipe(fin,
                         c.map(array_from_txt),
                         c.map(reshape(newshape=(1, -1))),
                         c.map(pca_obj.transform),
                         np.vstack)

print(components.shape)

(150, 2)
```

接着绘制出这些成分。

```
iris_types = np.loadtxt('data/iris-target.csv')
plt.scatter(*components.T, c=iris_types, cmap='viridis');
```

你可以验证一下，这种方法的结果与标准主成分分析方法（基本上）相同（比较图 A-2 和图 A-3）。

```
iris = np.loadtxt('data/iris.csv', delimiter=',')
components2 = decomposition.PCA(n_components=2).fit_transform(iris)
plt.scatter(*components2.T, c=iris_types, cmap='viridis');
```

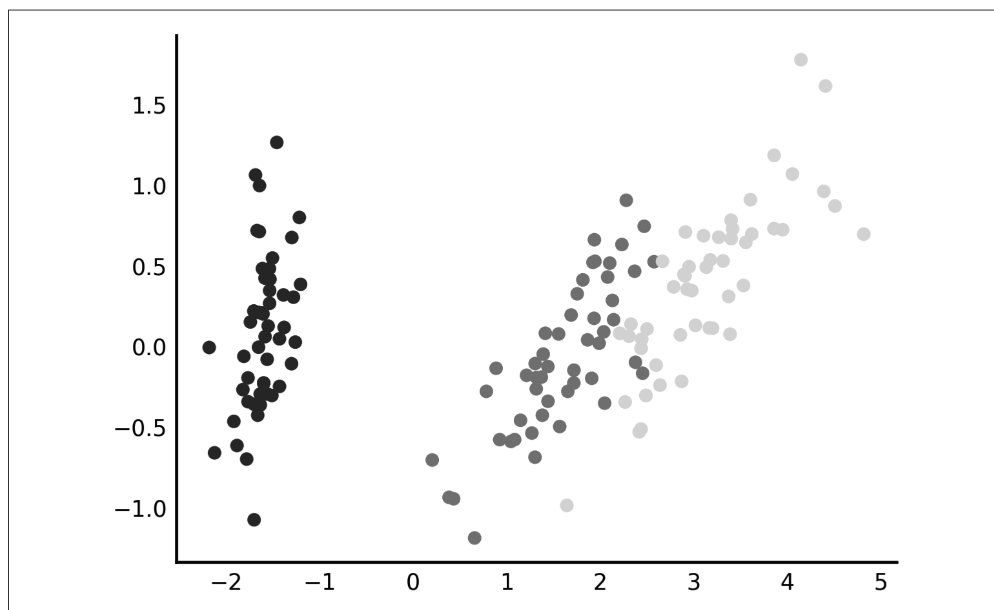


图 A-2：用流式主成分分析计算出的鸢尾花数据集主成分

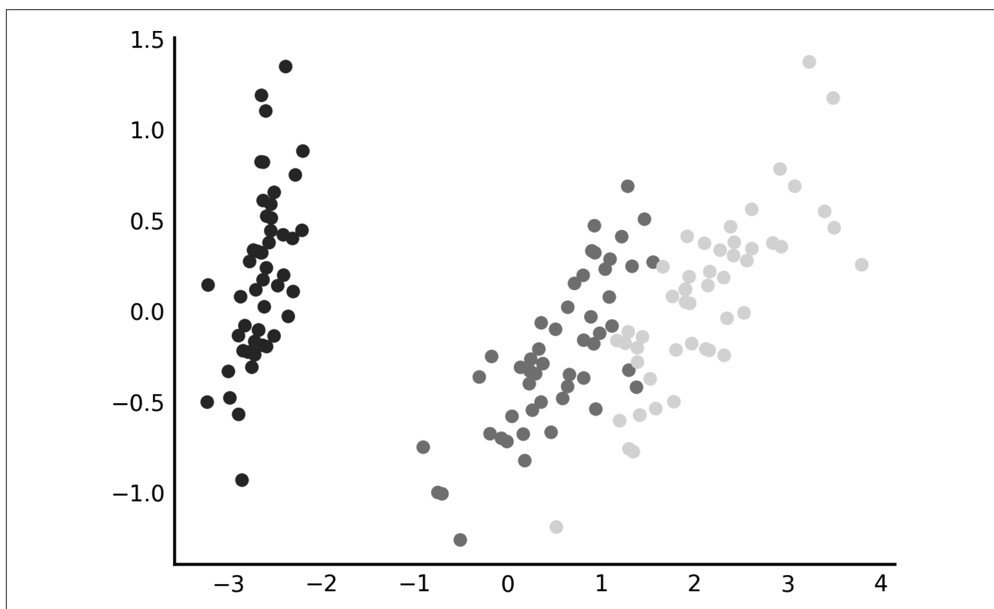


图 A-3：用标准主成分分析计算出的鸢尾花数据集主成分

当然，主要的区别在于流式主成分分析可以扩展到特别大的数据集。

A.20　答案：在管道开始位置添加一个步骤

这是“练习：在线解压”的答案。

可以将原始 `genome` 代码中的 `open` 替换为一个柯里化的 `gzip.open` 函数。`gzip` 中 `open` 函数的默认模式是 `rb`（read bytes），而不是 Python 内置函数 `open` 的 `rt`（read text），因此要指定这个参数。

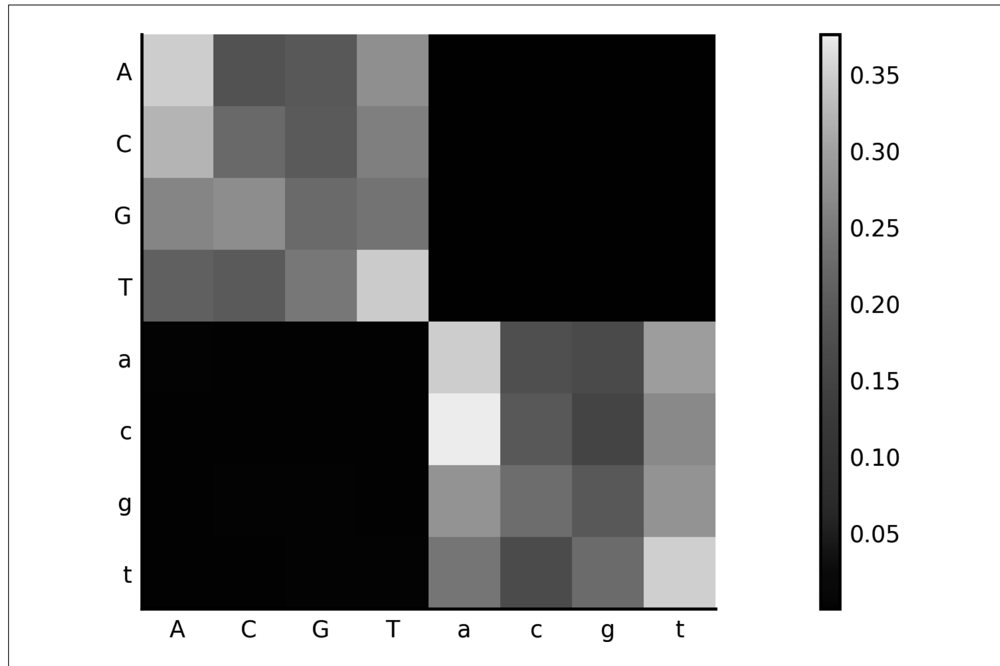
```
import gzip

gzopen = tz.curry(gzip.open)


def genome_gz(file_pattern):
    """Stream a genome, letter by letter, from a list of FASTA filenames."""
    return tz.pipe(file_pattern, glob, sorted, # 文件名
                   c.map(gzopen(mode='rt')), # 行
                   # 连接所有文件中的行
                   tz.concat,
                   # 去掉每个序列的标题
                   c.filter(is_sequence),
                   # 连接所有行中的字符
                   tz.concat,
                   # 去掉换行符和N
                   c.filter(is_nucleotide))
```

可以用压缩的果蝇基因组文件测试一下。

```
dm = 'data/dm6.fa.gz'
model = tz.pipe(dm, genome_gz, c.take(10**7), markov)
plot_model(model, labels='ACGTacgt')
```



如果想要一个特别的 `genome` 函数，那么可以定制一个 `open` 函数，根据文件名或使用试错法确定一个文件是否为 gzip 文件。

同理，如果你有一个全部由 FASTA 文件组成的 .tar.gz 文件，那么可以使用 Python 的 `tarfile` 模块代替 `glob` 来分别读取每个文件。唯一需要注意的是，必须用 `bytes.decode` 函数对每一行进行解码，因为 `tarfile` 返回的行是字节，而不是文本。

作者简介

胡安·努内兹－伊格莱西亚斯（Juan Nunez-Iglesias）是澳大利亚莫纳什大学的研究员，同时也是一名自由咨询顾问。此前，他在霍华德·修斯医学院的 Janelia Farm 研究所担任助理研究员（同 Mitya Chklovskii 一起工作），博士毕业于南加利福尼亚大学（专业为计算生物学，导师是 Xianghong Jasmine Zhou），并在该学校担任过研究助理。他的主要研究兴趣是神经科学和图像分析，并对生物信息学和生物统计学中的图形方法具有浓厚的兴趣。

斯特凡·范德瓦尔特（Stéfan van der Walt）是加州大学伯克利分校数据科学研究所的一名助理研究员，同时也是南非斯泰伦博斯大学的应用数学高级讲师。他致力于开源科学软件开发长达十余年之久，并乐于在一些会议和研习班中讲授 Python。斯特凡是 scikit-image 的创建者，同时也是 NumPy、SciPy 和 cesium-ml 的贡献者。

哈丽雅特·达士诺（Harriet Dashnow）是一名生物信息学家，曾在默多克儿童研究所、墨尔本大学生物化学系和维多利亚州生命科学计划项目中工作过。哈丽雅特在墨尔本大学获得了心理学学士、遗传学和生物化学学士以及生物信息学硕士 3 个学位，目前正在攻读博士学位。她在基因组学、Software Carpentry、Python、R、Unix 和 Git 版本控制等领域组织了一系列研讨会，并在其中讲授计算技能。

封面简介

本书封面上的动物是一只乐园维达鸟（学名 Vidua paradisaea），又称长尾维达鸟。这种体型小巧、类似麻雀的鸟类分布在东非，从南苏丹到安哥拉南部的地区。

乐园维达鸟的雄性和雌性通常难以分辨，直到繁殖季雄性长出繁殖羽才能看出来。繁殖期的雄性头部呈黑色，胸部呈褐色，后颈羽毛为亮黄色，腹部为白色。黑色尾羽长且宽，长度约为身体的 3 倍。

乐园维达鸟是绿翅斑腹雀的巢寄生者。雄鸟可以模拟雄性斑腹雀的叫声，因为它们的叫声更响亮，所以绿翅斑腹雀的养父母会给予它们更多关注。这种巢寄生本性使得它们很难被人工饲养，但在美国和其他一些国家中，雄鸟经常被当成宠物贩卖。长尾维达鸟被评定为无危物种。

O’Reilly 图书封面上的很多动物都是濒危物种，它们是这个世界的至宝。如果你想为保护动物贡献一份力量，请访问 animals.oreilly.com。

封面图片来自 *Wood’s Illustrated Natural History*。



微信连接



回复“数据科学”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

Python科学计算最佳实践：SciPy指南

本书结合数据实例，使用SciPy及NumPy、pandas、scikit-image等相关库，介绍了强大的Python科学计算工具，展示了如何使用Python编程来处理数据科学研究中可能遇到的现实问题，以及如何写出清晰优美的代码。

本书“麻雀虽小，肝胆俱全”，不仅探讨了作为计算工具本身的SciPy及其相关的库，还阐释了数据科学研究中一些必要的基础概念，是使用Python编程的数据科学研究人员阅读参考的理想选择。

- 探索Python科学应用的基础——NumPy
- 用NumPy和SciPy进行分位数标准化
- 图像区域网络及区域邻接图
- 频率与快速傅里叶变换
- 用稀疏坐标矩阵实现列联表
- SciPy中的线性代数
- SciPy中的函数优化
- 用Toolz在笔记本电脑上玩转大数据

“本书结合信号处理、图像处理、网络科学、生物信息学等领域的例子，展示了如何通过Python优雅地实现经典的算法，是一本难得的科学计算教材。”

——**Lav Varshney**
伊利诺伊大学厄巴纳-香槟分校

“生动展示如何通过Python的科学计算工具达到事半功倍的效果。”

——**Greg Wilson**
RStudio数据科学家
Software Carpentry联合创始人

胡安·努内兹–伊格莱西亚斯（Juan Nunez-Iglesias），澳大利亚莫纳什大学研究员，咨询顾问。

斯特凡·范德瓦尔特（Stéfan van der Walt），scikit-image的创建者，加州大学伯克利分校数据科学研究所助理研究员，南非斯泰伦博斯大学应用数学高级讲师。

哈丽雅特·达士诺（Harriet Dashnow），生物信息学家，曾在默多克儿童研究所、墨尔本大学生物化学系和维多利亚州生命科学计划项目中工作过。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／数据科学／Python

人民邮电出版社网址：www.ptpress.com.cn





ISBN 978-7-115-49912-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks